NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A281 038

|||||||||||||||||||||||||||

THESIS

IMPLEMENTING AN OPEN OCEAN THEATER IN NPSNET

by

James Harvey Covington Jr.

March 1994

Thesis Advisor: David R. Pratt
Co-Advisor: Anthony J. Healey

DTIC QUALITY INSPECTED 3

94-20398
|||||||||||||||||||||||||||

94   7   5   115

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
Implementing an Open Ocean Theater in NPSNET

5. FUNDING NUMBERS

6. AUTHOR(S)
Covington, James Harvey Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The problems addressed by this research were to establish an efficient set of data structures and functions to implement a realistic open ocean environment, to create a conceptual representation of the ocean surface that realistically animates waves in real time and coordinates the dynamic motions of simulated marine vehicles sailing on the surface, and to establish an object-oriented paradigm for the incorporation of graphical user interface (GUI) components into the present NPSNET structure.

The approach taken for this research was to develop a set of C++ classes that contained both the necessary data and methods to describe the ocean surface as a spatially organized hierarchy of dynamic geometric structures. The wave form associated with the surface was designed as a separate object to allow it to influence the periodic motions on surface marine vehicles as well as dictate wave height at any point and time.

The results of this work are the Ocean and Wave classes, an extension to the NPSNET Vehicle class, and the modification of an OSF/Motif application framework library that supports the implementation of an IRIS Performer simulation. The extensibility of the system is enhanced through the expanded use of C++ objects, which was proven by the successful integration of NPSNET into the Motif application framework.

14. SUBJECT TERMS
NPSNET, real-time, 3D, visual simulation, network, distributed, Performer, Motif, GUI, interactive, virtual world, vehicle dynamics, water waves

DTIC QUALITY INSPECTED 3

15. NUMBER OF PAGES
83

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

NSN 7540-01-280-5500

IMPLEMENTING AN
OPEN OCEAN THEATER
IN NPSNET

by
James Harvey Covington Jr.
Lieutenant, United States Navy
B.S., Florida State University, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1994
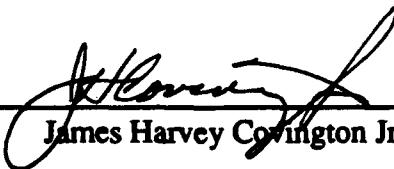
Author: _____
James Harvey Covington Jr.

Approved By: _____
Dr. David R. Pratt, Thesis Advisor
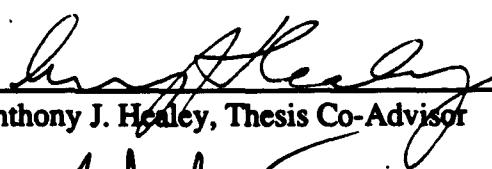
_____
Dr. Anthony J. Healey, Thesis Co-Advisor

_____
Dr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problems addressed by this research were to establish an efficient set of data structures and functions to implement a realistic open ocean environment, to create a conceptual representation of the ocean surface that realistically animates waves in real time and coordinates the dynamic motions of simulated marine vehicles sailing on the surface, and to establish an object-oriented paradigm for the incorporation of graphical user interface (GUI) components into the present NPSNET structure.

The approach taken for this research was to develop a set of C++ classes that contained both the necessary data and methods to describe the ocean surface as a spatially organized hierarchy of dynamic geometric structures. The wave form associated with the surface was designed as a separate object to allow it to influence the periodic motions on surface marine vehicles as well as dictate wave height at any point and time.

The results of this work are the Ocean and Wave classes, an extension to the NPSNET Vehicle class, and the modification of an OSF/Motif application framework library that supports the implementation of an IRIS Performer simulation.The extensibility of the system is enhanced through the expanded use of C++ objects, which was proven by the successful integration of NPSNET into the Motif application framework.

Accesion For		
NTIS CRA&I		☒
DTIC TAB		☐
Unannounced		☐
Justification _____		
By _____		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

I. INTRODUCTION

A. BACKGROUND

NPSNET is a real-time, three-dimensional (3-D), distributed interactive virtual world being continually developed by researchers and students in the Graphics and Video Laboratory of the Department of Computer Science at the Naval Postgraduate School. It began as a low-cost, workstation based simulator which used a locally developed network protocol and runs on commercially available Silicon Graphic Inc. (SGI) workstations [YOUN93]. In the versions that followed, added functionality and improved capabilites were added, resulting in a suite of complementary software applications that push forward the state-of-the-art in distributed virtual worlds. This research is a further effort in that direction.

B. MOTIVATION

1. The need for an open ocean environment

In the current implementation of NPSNET, the virtual battlefield is a limited area defined by a series of polygons that approximate the terrain. While this limited area is sufficient for land battles and the associated close air support, naval and air warfare takes place over a much broader area. Deep draft surface vessels and submarines, however, will operate in areas that are surrounded on all sides by many miles of water. In order for networked virtual world simulation to be an effective training and tactical development platform for naval personnel, the domain of NPSNET must be extended to include an ocean environment. The next logical extension to NPSNET is the addition of the littoral regions, followed by the open ocean. However, it makes sense from a design perspective to implement the open ocean first, since the interaction of waves and coastline is a conceptually more complex problem than the implementation of wind driven waves over deep water.

2. The importance of wave motion in visual simulators

In currently installed submarine attack center trainers and ship's periscope and navigation (SPAN) trainers, the ocean surface is modeled as a flat blue surface over which graphical images of ships are set in motion. Depending on the type of training desired, the submarine crew uses these images to practice anti-surface warfare (ASUW) periscope approaches, contact avoidance during inbound and outbound surface transits, or safe approaches to periscope depth. In those cases where the periscope window is relatively close to the surface (e.g. during an ascent to periscope depth), the motion of the seas is a large factor influencing the effective range at which the periscope officer can identify vessels which could present a collision hazard to the ship. It also plays a crucial role in the time required for the periscope officer to assess the number of reticle subdivisions occupied by a surface contact. This assessment, the key piece of raw data needed for visual range estimation, is more difficult to perform if the periscope window is rocking back and forth, and the reticle markings are rarely vertically aligned. The result of this difficulty is a trade-off between reduced range accuracy and increased contact observation time. Increased contact observation time is particularly undesirable in the ASUW approach scenario, since prolonged periscope exposure increases the possibility of counter-detection. These factors are not addressed in the visual simulation systems currently used. [COVI92]

From the surface anti-submarine warfare (ASW) perspective, a dynamic ocean surface is necessary to more accurately model both visual and radar detection of submarine periscopes. In a flat blue ocean simulation, visual identification of small foreign objects, e.g. a periscope mast, is unnaturally simple. Similarly in a computer-modeled radar, the absence of sea clutter caused by the reflection of radar waves off wave peaks will produce an artificially inflated accuracy of the detection and identification of periscope masts.

3. The importance of realistic vehicle motion

Marine vehicles, including both surface ships and surfaced submarines, maneuver in what is essentially a two dimensional world. They are restricted to operation at sea level,

2

minus the draft necessary to establish neutral buoyancy. Any excursions above or below this level will result in a net force on the vehicle restoring it to its original level. To implement realistic marine vehicle motion, however, these periodic excursions above and below the neutral buoyancy level must be modeled in a graphical simulation. Changes in the vehicle's orientation in the world, resulting from such effects as pitch and roll, must also be modeled in order to achieve visual realism. To a large extent, these rotational and vertical motions are closely linked to the motion of the ocean surface itself. Therefore a computer model which implements wave dynamics should also be applicable to more realistic ship dynamics.

By incorporating a dynamic ocean surface in NPSNET, ship and submarine crews can supplement their training initially with an off-the-shelf computer hardware platform, and later incorporate this same level of realism into existing trainers by means of hardware upgrades and software modifications.

C. OBJECTIVES

The objective of this research is to design an open ocean environment utilizing the Silicon Graphics Iris Performer [SGI92] visual simulation toolkit, and incorporate this environment into NPSNET IV, the networked virtual world project currently under development at the Naval Postgraduate School. To achieve this objective, the environment should consist of, as a minimum, a dynamic model of the ocean surface with user control over essential wave parameters such as wavelength, amplitude, and direction. It may also allow the projection of varying texture images consistent with the sea state being modeled. Furthermore, the current NPSNET vehicle class hierarchy must be extended to provide for the control of ship motion in a manner consistent with the generation of the moving ocean surface.

A secondary objective of this research is to provide a framework for the incorporation of essential graphical user interface (GUI) components for both this application and future modifications to NPSNET. By constructing an object-oriented framework for the addition

of user-interface components, development time spent by future researchers can be made more productive. The current implementation of NPSNET, written in C++, utilizes object-oriented techniques in the control of virtual world objects such as vehicles, weapons, and stationary objects [ZYDA93]. This concept will be extended to allow the creation of basic program control objects that drive the overall simulation.

D. ORGANIZATION

The previous sections of this chapter have stated the objectives and motivation for providing an open ocean environment for NPSNET. Chapter II provides an overview of the NPSNET project, including its purpose, history, general program design, and current research being conducted. Chapter III discusses the design considerations of the dynamic ocean surface, including an overview of applicable wave theory, survey of earlier work, and computational considerations. The design of a realistic model of marine vehicle dynamics is detailed in Chapter IV, including a discussion of applicable theory, earlier work, and alternative representations. Chapter V gives a brief overview of the Motif graphical user interface (GUI). Next, Chapter VI presents the design of a generic application framework which runs Performer within a Motif-based application, while Chapter VII details the implementation of this research into the NPSNET project. Chapter VIII presents the conclusions reached during the course of this research, including performance analysis, advantages, limitations, suggestions for software improvements, and areas for suggested further research. Finally, the appendices contain the source code for the Performer extensions to the Motif application frameworks library.

4

II. OVERVIEW OF NPSNET

A. PROJECT PURPOSE

In 1990, students in the Naval Postgraduate School's (NPS) Department of Computer Science, working in the school's Graphics and Video Laboratory, began a project known as the Naval Postgraduate School Networked Vehicle Simulator (NPSNET). NPSNET is a low-cost, real-time, networked vehicle simulator which runs on the Silicon Graphics IRIS family of graphics workstations [ZYDA93]. Its primary purpose is to provide a means through which the NPS students, primarily U.S. military officers, can experience the potential benefits of networked virtual environments on the tactical training of today's and future military forces, as well as participate in its development. Students participating in this research gain the experience of working in a large project and learning where major difficulties in software evolution are, while contributing to the state of the art in virtual environment software technology. At future commands, these students will be intimately aware of both the benefits and limitations of networked virtual environments.

B. GENERAL PROGRAM DESIGN

NPSNET's source code is written in C++ to take advantage of the object-oriented paradigm offered by C++ classes. Utilizing this paradigm, virtual world objects such as vehicles and weapons are implemented in a class hierarchy that provides for inheritance of properties from more abstract classes. In this way, for example, a tank object and a jeep object can share the properties belonging to all land vehicles, such as the inability to float or fly, while specializing in those areas that they alone possess, such as maximum velocity, or the existence of a gun turret. This object-oriented design also serves to ease the process of expanding the virtual environment to include a wider variety of vehicles, and other objects.

NPSNET follows the general conventions of a Performer application as seen in Figure 1. During the initialization phase, a configuration file is read which specifies any

desired configuration parameters, such as window size, multiprocessing mode, frame rate, multisampling, etc. The configuration file also specifies file names that will be read to form the visual database and provide network and sound support.

NPSNET generally runs within a single channel, but has added the feature of a "video missile" which shows the missile's view of the scene as it flies toward its target. This effect is accomplished by the addition of a second channel to the single graphics pipeline, forming a "picture within a picture." The channel configuration also includes the initialization of such channel-dependent features such as a pfEarthSky, pfFog, and pfLight.

The pfScene in NPSNET is composed of many elements, the number and composition of which will vary during the execution of the program. As players join and leave the network, for example, the local application must add their graphical icons to the local scene. To avoid any unnecessary delays in the local application, all defined models are loaded at this time, and instances of these models are "cloned" when needed. NPSNET also uses this portion of the program to perform network initialization, and the pre-loading of texture image data. The time required to perform this initialization is considerable; approximately seventy two seconds elapse from application start to the first drawn frame. To provide a level of user confidence that the application has not "locked up" during this initialization phase, a call to pfSync is made, which allows the forked processes to start up, and a scripted sequence of opening credits and program status reports is generated by the draw callback function.

In addition to the standard Performer structure outlined above, NPSNET forks two additional processes to run in parallel with the Performer application, the Flight Control Stick (FCS) input process and the network communications process. A more detailed description of NPSNET can be found in [PRAT93][ZYDA93]

C. RECENT RESEARCH

Research associated with the NPSNET project is as varied as the backgrounds of the students involved, which span multiple armed service branches, and multiple warfare

specialties within each branch. One recent accomplishment has been the development of real-time environmental effects such as smoke plumes, fire, and dust trails [CORB93]. Another has been the redesign of the NPSNET network interface to operate with standard Distributed Interactive Simulation (DIS) protocol data units (PDU's) versus an original local architecture [ZESW93]. The immersive nature of the NPSNET virtual world (VW) has been further enhanced by the development of the NPSNET Spacialized Sound Server, providing spatialized aural cues to events in the VW.

As NPSNET research continues, the incorporation of autonomous forces into the VW continues to be a vital issue. Computer-driven players, originating from workstations communicating over the network, use scripted paths, collision-avoidance, and rule-based algorithms to interact with the human players in the VW. Work continues in this area to provide players whose behavior forms an acceptable substitute for an equal number of human players, thus adding to the believability of the simulation. Other areas of research involve improvements to the computer-human interface, such as the potential of incorporating a head-mounted display (HMD), and the recent inclusion of a stereoscopic display using the Stereo-Graphics CrystalEyes system.

```
┌─────────────────────────┐
│  Performer Initialization│
└─────────────────────────┘
              │
              ▼
┌─────────────────────────┐
│  Allocate shared memory  │
│  for forked processes    │
└─────────────────────────┘
              │
              ▼
┌─────────────────────────┐
│  Fork off separate       │
│  application, cull,      │
│  and draw processes      │
└─────────────────────────┘
              │
              ▼
┌─────────────────────────┐
│  Set up application      │
│  configure pipes,        │
│  load database           │
└─────────────────────────┘
              │
              ▼
┌─────────────────────────┐
│  Main simulation loop    │
│  update viewpoint        │
│  compute DCS's           │
│  update dynamics, etc.   │
└─────────────────────────┘
              │
              ▼
┌─────────────────────────┐
│        pfExit            │
└─────────────────────────┘
```
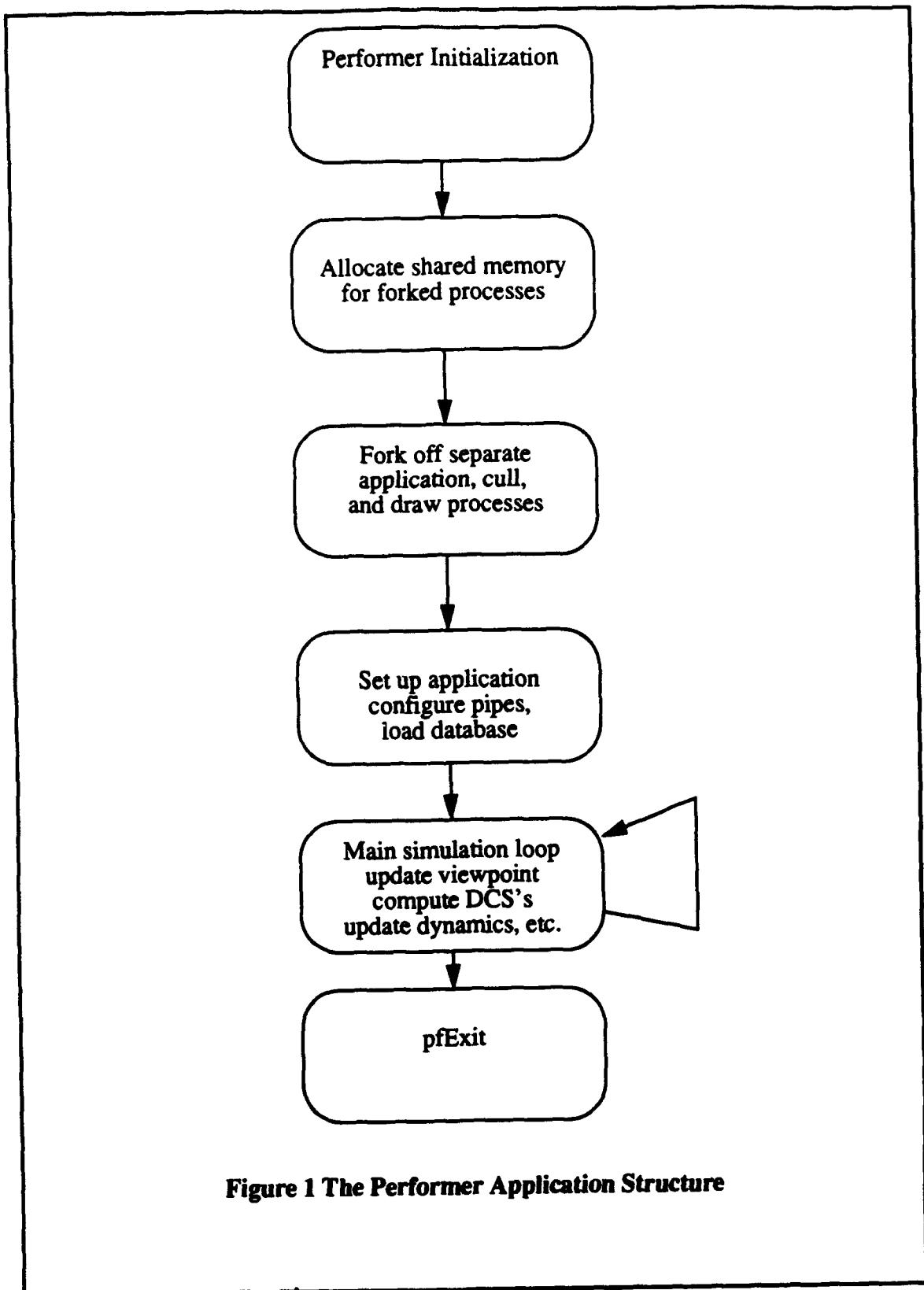
Figure 1 The Performer Application Structure

III. OCEAN DESIGN CONSIDERATIONS AND IMPLEMENTATION

A. OVERVIEW OF WAVE THEORY

The study of waves is an extremely complex topic, as wave phenomena take on such diverse forms as the ripple a thrown pebble makes in a still pond, the wake of a ship, or the white caps generated by a storm at sea. These phenomena are all examples of surface waves, and as such influence the appearance of the ocean surface, and the behavior of marine vehicles on the surface. Other wave phenomena include internal waves, whic found in areas where water density changes drastically, such as at thermal layers. Ano category of waves is the inertial wave, which is formed from the earth's rotation, in much the same way that winds are generated in the atmosphere [EARL84]. These waves generally have very long periods (several minutes in duration), and as such can be ignored for the purpose of marine vehicle dynamics [NEWM77].

Surface waves themselves can be described as having both linear and nonlinear behavior. The periodic ocean swells resulting from the gradual buildup of wind-driven wavelets can be accurately described by simple periodic functions, i.e. sinusoids or trochoids [BLAG62]. Breaking waves, such as those near shores and reefs, or those caused by the motion of a ship's bow through the water, demonstrate the nonlinear effects also present in surface waves [PEAC86]. In open ocean applications the effect of nonlinear wave behavior on large draft vessels can be neglected. These waves can be later approximated through visual means only, such as adding a noise texture.

Waves are found in many areas of physics; sound waves, light waves, seismic waves, etc. [HALL81]. But surface water waves possess a unique characteristic, in that the water particles themselves travel in closed, or nearly closed orbits. The observed propagation of waves is merely a propagation of wave shape resulting from the timing of these individual orbits, as can be seen in Figure 2.
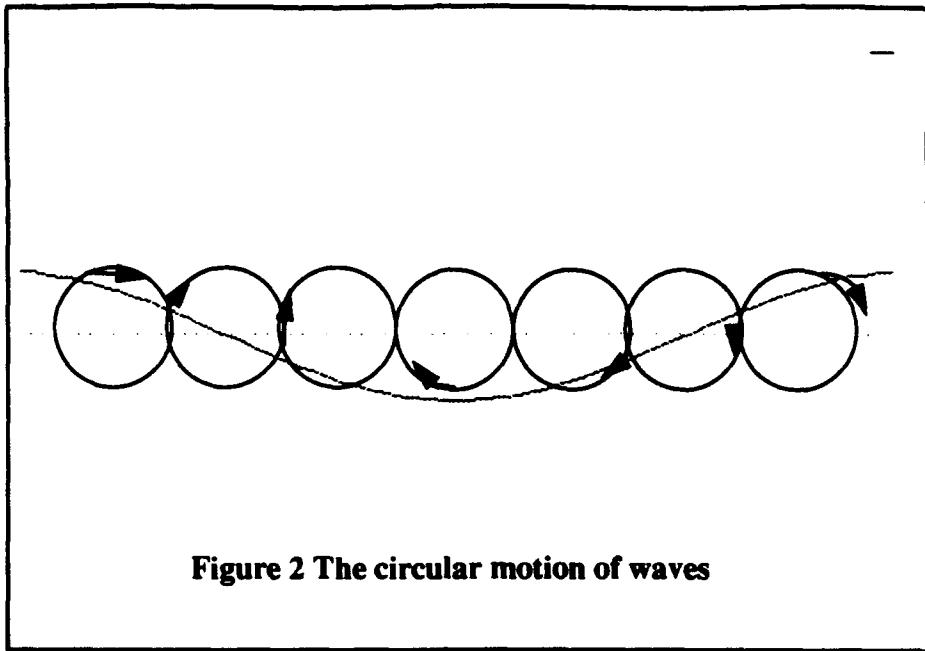
Figure 2 The circular motion of waves

The observed shape of many large ocean waves, those with a long shallow trough and a sharply defined peak, are best approximated by a trochoid or cycloid. A trochoid is the shape described by the path a point on a disk would trace through space, while the disk is rolled along a flat surface. If the point lies on the perimeter of the disk, the peak exhibits a discontinuity in its slope, and the curve is referred to as a cycloid. Trochoids and cycloids are described mathematically by a pair of parametric equations, in which the x and y variables are defined as a function of a third variable t. The basic cycloid equation is:

$$x = k(t - \sin(t)) \qquad \text{(Eq 1)}$$
$$y = k(1 - \cos(t)) \qquad \text{(Eq 2)}$$

The basic waveform resulting can be seen in Figure 3. A simple inversion of the second equation, and adjustment to the parameter k, can form a realistic representation of water waves, as shown in Figure 4.
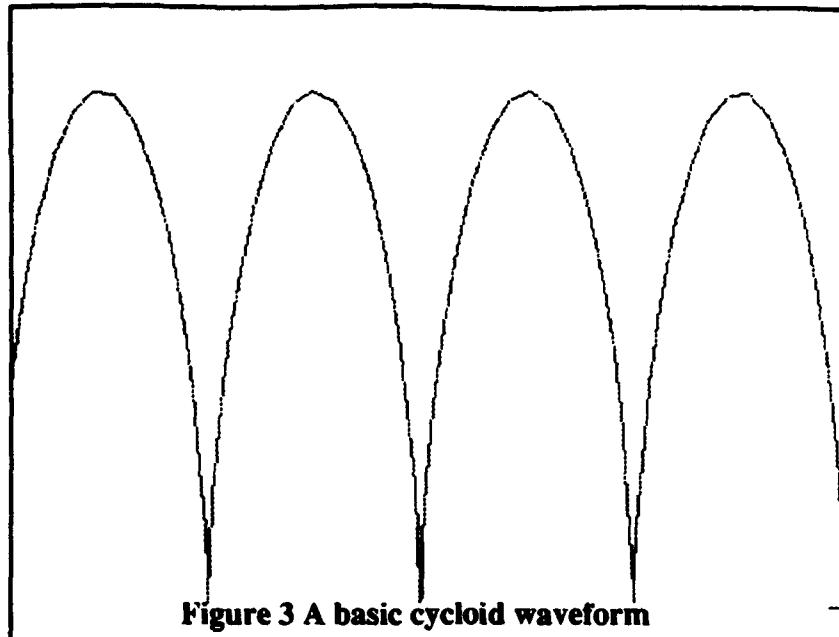
Figure 3 A basic cycloid waveform
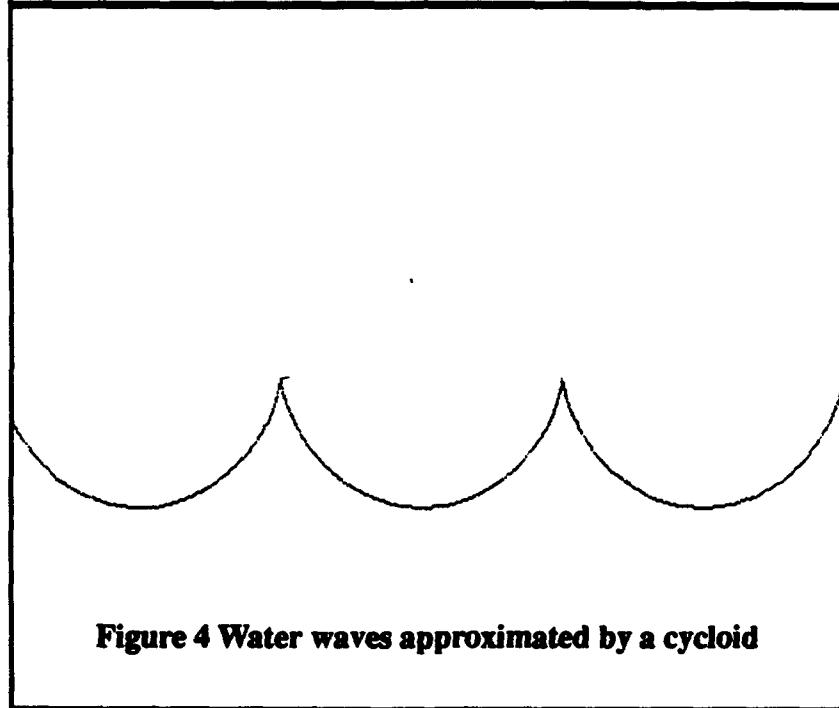
Figure 4 Water waves approximated by a cycloid

Since the primary surface waves encountered by and affecting marine vessels are those generated by winds, it is beneficial to review the process of ocean wind wave generation. Wind currents generate variations of air pressure over the surface of the water, resulting in "ripples" or "wavelets." As these wavelets are generated, they present a surface

against which the winds can directly impinge. This impingement of the wind against the sides of the wavelets increases the rate of energy transfer from the wind to the water, thus accelerating the growth of the wave. Wave height will continue to grow until the ratio of wave height to wave length reaches a value of approximately one to seven. When this latter condition is reached, the individual waves will break, forming white caps [EARL84].

A single wave component can be fully described through the parameters of amplitude, wave length, period, phase angle, and direction. The starting equation then for a sinusoidal wave component is:

$$\zeta = \zeta_a \sin k \left(x - V_w t \right) \qquad \text{(Eq 3)}$$

where ζ is the value of the wave function, ζ_a is the wave amplitude, $k = 2\pi/L$ is the wave number, L is the wave length, x is the horizontal distance from the wave origin, V_w is the wave velocity, and t is the time variable. For ocean waves, the wave velocity is a function of the wave number and the water depth h given by

$$V_w = \frac{g}{k} \tanh (kh) \qquad \text{(Eq 4)}$$

When the water depth becomes sufficiently large, the tanh function approaches 1, and the wave velocity essentially becomes a function of the wave number, and therefore of the wave length. In these terms we get:

$$V_w = \frac{gL}{2\pi} \qquad \text{(Eq 5)}$$

Therefore a complete description of a single wave component can be made by specifying only the amplitude, wave length, and direction [BHAT78].

In the oceans, the appearance of the sea state is described not by a single sinusoidal wave component, but by the sum of many individual wave components of varying amplitudes, wavelengths, and directions. And yet careful observation over time reveals an average wave height, and general direction of the seas, which has a direct bearing on piloting decisions. The implementation of a computer model of the ocean surface should

12

be able to generate the defining parameters for each wave component, based on a desired overall amplitude, wavelength, and direction. One way to accomplish this is to declare an arbitrary average wave height, wave length, and direction, then generate the parameters for the individual wave components by some reasonable weighting function. The result of the weighting function should be that the amplitude of an individual wave component be maximum when the direction of the wave is coincident with the desired direction of the seas, and should be minimum when perpendicular. Wavelength may be computed in a similar manner.

There is an extensive body of knowledge regarding the theory of ocean wave generation and propagation, and a number of models and waveforms that describe and predict ocean wave behavior. The motivation for this research, however, does not lie in the accurate scientific visualization of ocean waves, which at present is the realm of massively parallel supercomputers. Rather, the goal of this research lies primarily in obtaining a model for ocean wave generation which is reasonably realistic, yet falls within the computational constraints of real-time applications. For this reason we model the ocean surface as a sum of simple sinusoidal waves.

B. SURVEY OF EARLIER WORK

There are several examples of research in which ocean waves have been modeled in computer graphics. In one early effort the ocean surface was generated by raytracing a complete height field [MAX81]. Noise texture mapping has also been utilized to describe wave trains [PERL85]. In this technique, band-limited noise was summed up, thus comprising relatively complex patterns of waves and ripples. In both of these cases however, the appearance is only valid from a significant height above the surface. For the purpose of this research, the model must be realistic when viewed from a position close to the water surface.

In 1986, Peachey developed a model of ocean waves which approached and broke upon a shore. This model was rendered using an off-the-shelf scan-line rendering program.

Peachey's model approximated the cycloidal nature of natural ocean waves by representing shallow waves with a sinusoidal function and steeper waves with a quadratic function, providing a linear blending between the two. This model also took into account the varying depth of the water as the waves approached an irregular beach, and provided a good visual effect of spray through the use of a particle system. The images produced using this model took approximately one hour of CPU time on the equivalent of a VAX 11/785 FPA. Included in this time was approximately two to five CPU minutes for the generation of the phase function table for each of three wave components [PEAC86]. Another limitation of this model, besides the fact that it could not produce real-time animation, is the fact that each wave component was fixed in amplitude. This was not a problem in Peachey's effort, however, since the end goal was a computer-generated animation sequence, not a real-time virtual world.

Also in 1986, Fournier and Reeves presented a model for ocean waves based on the Gerstner, or Rankine model, in which the particles of water describe circular or elliptical orbits. Again the depth of the water is taken into account and allows for refraction of waves as well as the formation of breakers. The Fournier and Reeves model mathematically describes the circular orbits of individual water particles using a variation on the cycloid parametric equations presented earlier. Modifications were made to model the effect of wind on the crests of the waves, and to impose a degree of the randomness found in the real sea. The ocean surface was graphically modeled as a set of bicubic patches. Again, this model was not meant to approach real-time graphics speed, as the rendering of these patches in a single frame took as little as two to as many as ten hours on a Computer Consoles Power 6/32 computer [FOUR86].

C. VIRTUAL WORLD SIZE VERSUS POLYGON COUNT

In every field of research, decisions must be made which involve trade-offs between two or more desired goals. In the field of economics this is referred to as the opportunity cost of a capital expenditure. And in computer science there have been several such trade-

offs, e.g. memory cost versus execution speed. In the field of computer graphics in particular, one of the key issues has been image quality versus execution speed. At one extreme is the decision to render images using ray tracing algorithms; the images generated by ray tracing have an extremely high image quality but can easily take several minutes to many hours to render, depending on the detail desired. Obviously this extreme is unacceptable for real-time graphics applications, given present hardware limitations. The goal then is to produce the highest quality image that can be updated as necessary and rendered at an acceptable frame rate.

The terrain database hierarchy used in NPSNET IV is predicated in part upon the premise that the shape of the terrain is random and fixed in time. In the case of hills, valleys, and other geological features this assumption is valid over the duration of a real-time application. Varying the level of detail of such a varied visual database in order to minimize the number of polygons being sent down the graphics pipeline has been a major accomplishment in previous research on the NPSNET project. Level-of-detail nodes (pfLOD) are provided as part of the Performer toolkit to facilitate this effort, so that in a properly organized database, the number of polygons required to generate objects or terrain decreases in inverse proportion to the distance from the view point. In actuality, the level-of-detail provided by pfLOD's does not vary continuously, but consists of a finite set of pfLOD's each of which has a specified range. Provision is made for overlap between adjacent pfLOD's to avoid image discontinuity.

When designing an ocean database, this premise of fixed and random features is no longer valid. Instead, the ocean surface can be described at any point in time based upon the sum of the individual wave components defined for the area. Therefore the only predefined database for the ocean is a mesh of polygons whose vertices each define the wave height at that particular point on the surface.

D. THE OBJECT-ORIENTED OCEAN DATABASE

1. The N-ary tree

The final design implementation of the ocean was accomplished using an object-oriented hierarchy. The top level of this hierarchy, designated as the OceanMaster class, encompasses a pfSwitch, and contains three instances of the Ocean class. The Ocean class provides a single class definition for all Performer nodes below the pfSwitch declared in the OceanMaster class. The Ocean class has two constructors, one public and one private. Both constructors are designed so that they recursively subdivide the ocean surface, first horizontally and then vertically, until the angular spacing is such that a single geoset can be constructed. The public constructor takes care of allocating storage for, and computing the coordinates of the vertex, texture, and normal arrays. Pointers to these arrays are passed as arguments to the private constructor, so that a single vertex array is allocated for the entire Ocean class hierarchy.

The effect of this organization is such that an n-ary tree is constructed, with n being defined as a constant in the newOcean.h file. As currently implemented, n is defined as six. The various instances of the Ocean class can be seen visually in Figure 6. The spatial organization achieved by this recursive subdivision is similar to the quadtrees used in the NPSNET terrain database [MACK91], and has the same goal of improving the culling efficiency of the Performer simulation.

The vertices defined for the ocean database are defined such that all vertices lie on a fixed number of vectors extending from the origin, with the vectors themselves spaced equally around the origin. In this manner the number of vertices which subtend the horizontal field of view is constant, regardless of distance from the viewpoint. Furthermore, the spacing of vertices along a single vector is computed in such a fashion as to ensure that, at a predetermined height-of-eye, any two adjacent vertices on a single vector will subtend a constant vertical angle as well. Figure 5 shows this arrangement from a view directly overhead. Note that the polygons cover a progressively larger surface area as distance from

the origin increases. A continuously varying level-of-detail is accomplished without the use of pfLOD's. Also note that the radial lines extending from the origin provide continuity for the polygons as the level-of-detail progressively decreases. There is no need in this design for seam-stitching or added polygons at level-of-detail transition points. The effect of constant angular spacing is evident in Figure 7, which shows the view from the nominal height of eye, looking toward the horizon with a -20 degree pitch.

Figure 5 Wireframe view of ocean surface from directly overhead

1st level Ocean object
hangle = 60
vangle ~= 90

2nd level Ocean Object
hangle = 30
vangle = 90

3rd level Ocean Object
hangle = 10
vangle = 15

Individual pfGeode

Figure 6 The Ocean Class Spatial Hierarchy
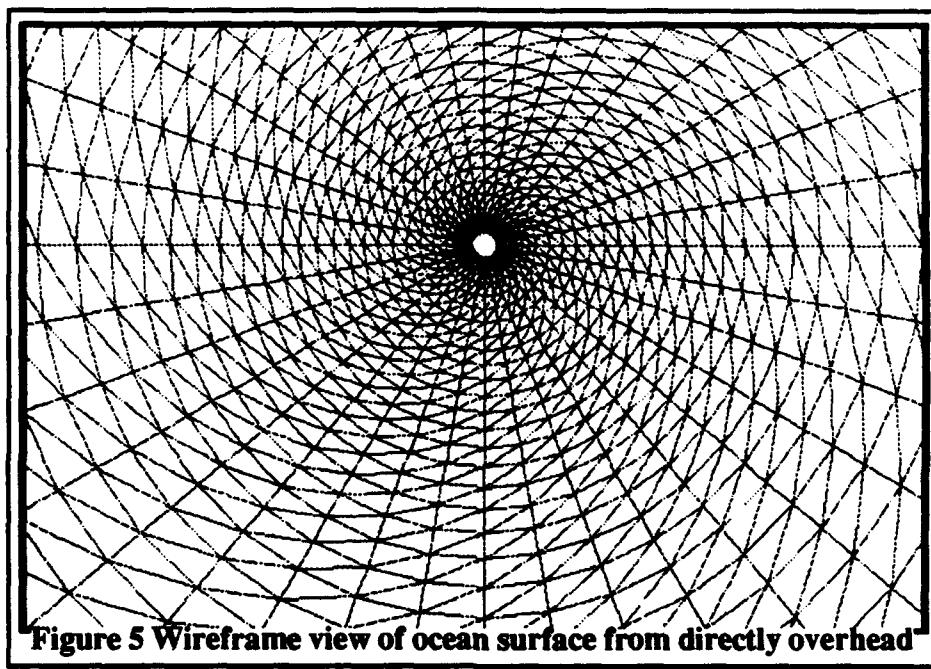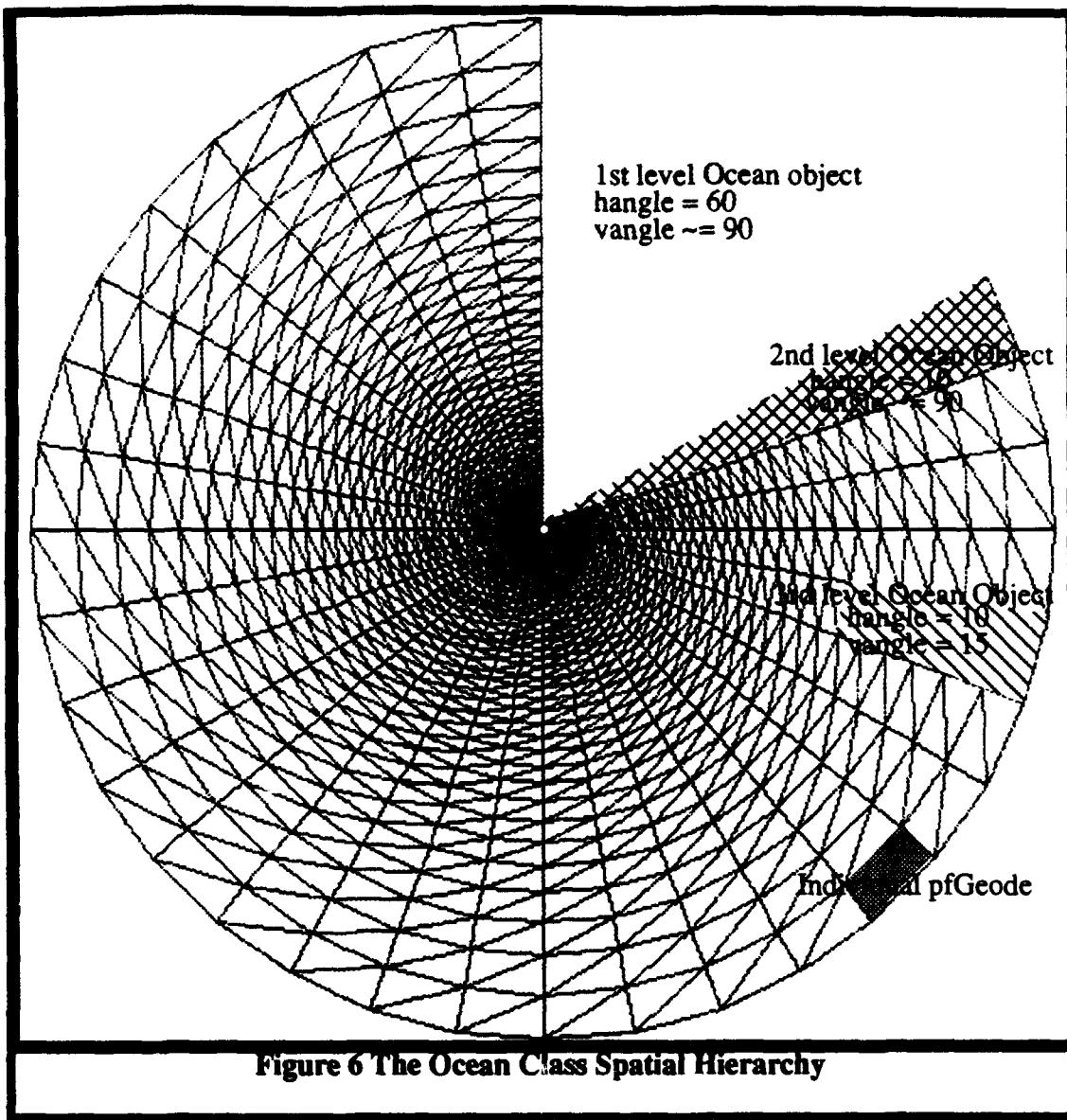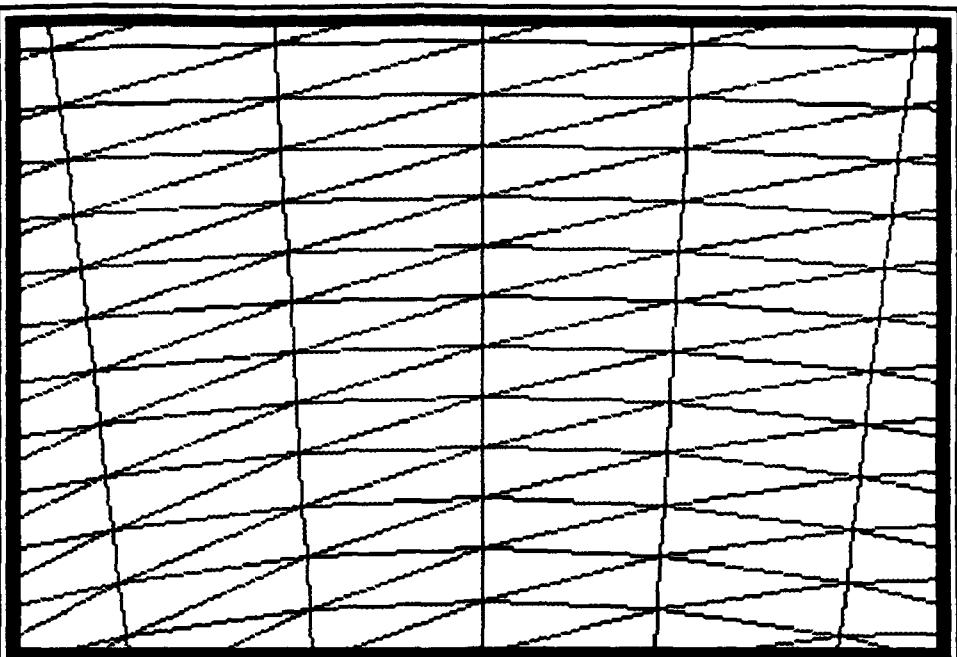
Figure 7 Wireframe view of ocean surface at nominal height of eye, -20 degree pitch angle

2. Multiprocessing considerations

As was mentioned above, the OceanMaster class contains three instances of the Ocean class, each of which is connected to the visual database by means of a pfSwitch. This creates a three element buffer to prevent inadvertently altering the data that is currently being used for rendering by the draw process. In the default multiprocessing mode, the pfConfig function causes the application to fork off separate application, cull, and draw processes which operate on the visual database in a pipeline fashion, i.e. while the application process is working on frame n, the cull process is working on frame n-1, and the draw process is rendering frame n-2. Following this convention, the application process needs three buffers; it will be processing the '(n+2) mod 3'th buffer at the same time the draw process is rendering the nth buffer. The only methods needed for the OceanMaster class are the animate() method, which selects the next buffer to animate and updates the active child of the pfSwitch upon completion, and the predraw and postdraw callbacks,

which are used to ensure that the application does not attempt to animate the child of the pfSwitch which is currently being drawn.

3. The Wave class

The waveform definition, which describes the appearance of the ocean surface at any instant in time, is implemented as a separate object, a global instance of the Wave class. As was seen above, due to multiprocessing considerations inherent in the Performer application paradigm, three instances of the Ocean class are instantiated in a buffer arrangement. There is no need to instantiate three instances of the Wave class, however. Although the three Ocean buffers may differ in their height values due to the time differential between successive calls to their animate() methods, the parameters defining the basic waveform do not vary, and thus one instance will suffice. Taking advantage of the fact that only one instance of the Wave class will be instantiated in a single application, a global pointer to the Wave class is defined along with the Wave class definition itself, "theWave." As long as the Wave object is created by the application prior to the OceanMaster object and any other objects that might need to access it, this arrangement works quite well.

The Wave class can be instantiated to be composed of any discrete number of separate wave components. A single direction, amplitude, and wavelength are also provided as arguments to the constructor. The algorithm utilized by the update_wave_parameters() function then takes these values and constructs the individual values for each wave component. The logic is such that the wave components are distributed evenly in the $\pm 90°$ range about the axis of the specified direction. The angular separation between the direction of a single wave component and the overall direction axis is used to compute a weighting factor which is then used to adjust the amplitude of the component. The end result is that the wave component whose direction has the greatest angular displacement from the desired direction will also have the smallest amplitude. To simplify the process of summing individual wave components, each component is further

broken down into its two component vectors. When the wave height is computed for a specific x-y coordinate, for example, the x value is considered first, its displacement from the y-axis used to generate a sum of all the wave component vectors which lie along the x-axis. The same is done for the y value, with the resultant sum being the total wave height at that point for the given time. An example utilizing this method for three wave components is given in Figure 8.

Number of waves: 3
Amplitude: 15.0
Wavelength: 600.0
Direction 60.0

Wave 0
Amplitude: 5.0
Direction 60.0

Wave 1
Amplitude: 3.53
Direction 15.0

Wave 2
Amplitude 3.53
Direction 105.0

Wave 0- x component
Wavelength 300.0
Wave velocity 21.64

Wave 0- y component
Wavelength 519.6
Wave velocity 28.48

$$waveHt(x, y) = \sum_{i=0}^{numWaves} XwaveHt_i(x) + \sum_{i=0}^{numWaves} YwaveHt_i(y)$$
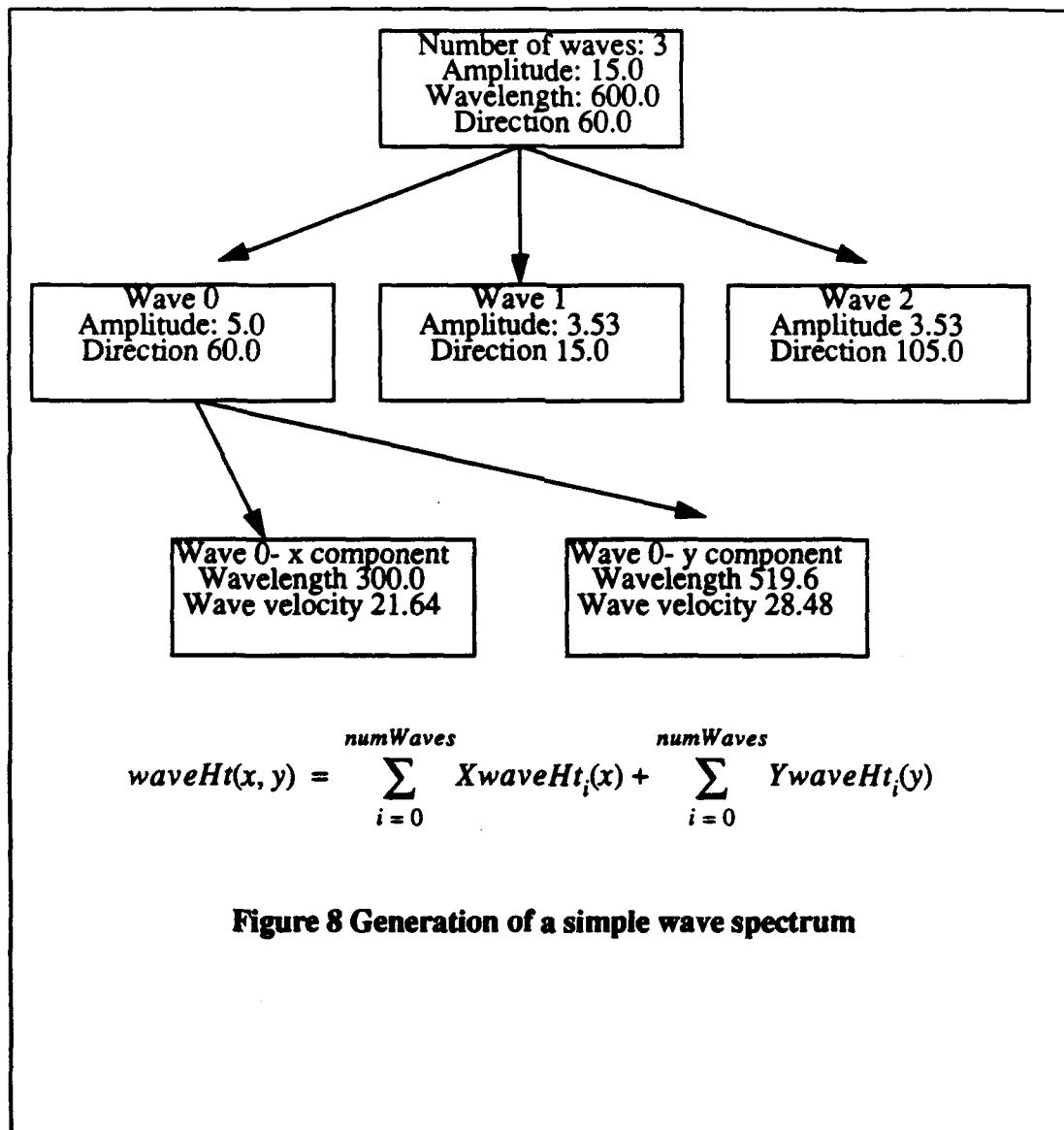
Figure 8 Generation of a simple wave spectrum

The decision to model the wave form as an object separate from the ocean itself deserves further justification. The same wave equations which define the motion of the ocean surface also serve to define the periodic motions of the marine vehicles which sail on the surface. The theory behind this motion and the utilization of the Wave class in predicting it are discussed in the following chapter.

IV. SHIP MOTION

A. OVERVIEW OF MARINE VEHICLE DYNAMICS

The field of modern marine vehicle dynamics goes back to the year 1737, where it has its roots in Euler's theory of motions in still water [BLAG62]. In his theory, Euler likened the oscillatory motion of a floating vessel to a pendulum, ignoring any effects of water resistance or other external forces acting on the vessel. His findings were published in the text *Ship Science* in 1749. In this same period, such noted mathematicians as Poisson and Bernoulli also contributed to the theory of oscillatory motion of floating vessels. Other noted pioneers in this field include Froude, Krilov, and Joukowsky [BLAG62].

As is true in the kinematics of any rigid body, marine vehicles experience six types of motion, three translational and three rotational. In the following discussion, a rectangular earth coordinate system is used, in which the x axis points north, the y axis points east, and the z axis points down. A body coordinate system for the vessel is also used, in which the x-axis runs longitudinally down the ship, the y-axis runs athwartships, and the z-axis runs up, as shown in Figure 9.
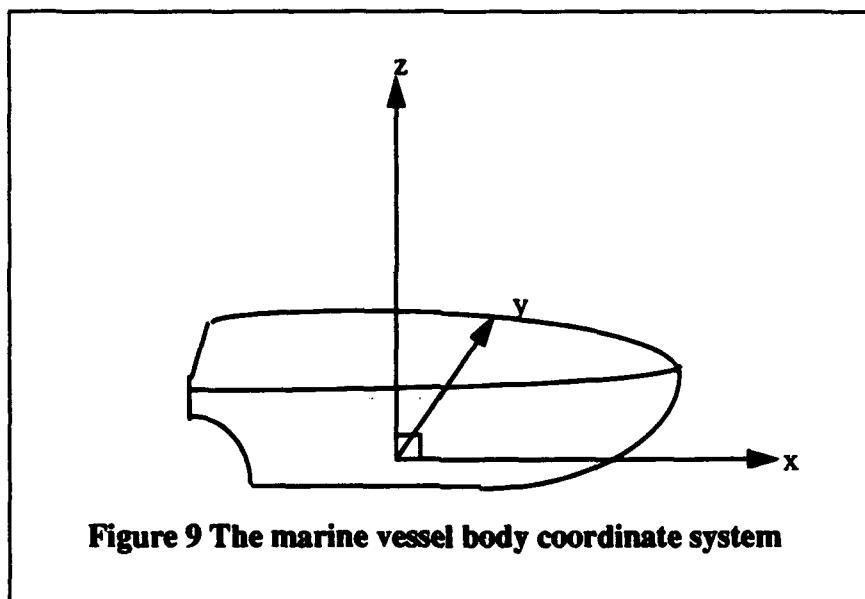
Figure 9 The marine vessel body coordinate system

In determining the position of a ship, mariners typically consider the latitude, longitude, course, and speed to be a complete description. In the rectangular coordinate system being used, the latitude would correspond to the x-coordinate, the longitude the y-coordinate,[1] the course the rotation about the z axis, and the speed a rate of change of the x-y coordinate values.

It is apparent to anyone who has watched a ship moving through the water that these are values do not provide a complete description of a ship's position at any instant of time. Moving ships are almost always in some sort of oscillatory motion as well as sustained motion over ground, and these oscillatory motions are comprised of all six types:

Surge - motion forwards and backwards in the direction of travel[2]

Sway - athwartship motion

Heave - vertical motion

Roll - angular motion about the longitudinal axis of the ship

Pitch - angular motion about the transverse axis

Yaw - angular motion about the vertical axis

The theory underlying these oscillatory motions is extensive and is thoroughly discussed in various texts [NEWM77][BHAT78][BLAG62][DEBE57]. No attempt will be made here to duplicate that effort. In order to lay the foundation for the implementation of a ship model, however, a brief summary is in order.

1. Encounter frequency

The waves in the ocean travel with a particular frequency relative to the earth. As a ship travels through the water, the rate at which it encounters the waves depends on its speed and course relative to the seas. If the seas are directly abeam, the encounter

---

1. This analogy is used for illustrative purposes only. It is not completely accurate, since latitude and longitude are, of course, angular measurements. Degrees of latitude can be translated directly into a linear displacement (1 degree is approximately 60 nautical miles), while the linear displacement associated with the degrees of longitude varies with latitude.

2. The surge, sway, and yaw components are not purely oscillatory, since they lack the inherent restoring forces provided by gravity and buoyancy. If the exciting forces causing the linear motion acts alternately from opposing directions, however, the motion will be oscillatory [BHAT78].

frequency, referred to mathematically as ω_e, is the same as the wave frequency. If the ship is driving through head seas, the encounter frequency is greater than the wave frequency. And lastly, if the ship has following seas, the encounter frequency is less than the wave frequency; indeed, it may be in the reverse direction of the seas, if the ship is overtaking the waves. Encounter frequency is given by

$$\omega_e = \omega_w\left(1 - \frac{\omega_w V}{g}\cos\mu\right) \qquad \text{(Eq 6)}$$

where V is the ship's speed and m is the encountering angle, measured clockwise from the direction of wave travel to the direction of ship's heading [BHAT78].

2. Heaving

Consider for a moment, the heaving motion of a ship. If the ship is somehow forced deeper into the water, the buoyant force on the ship will now be greater than the force of gravity, resulting in a net force upward after the initial exciting force is removed. The ship will now be accelerated upward back to its equilibrium position. However, the ship's inertia will carry it past this point and cause it to experience a net force acting downward. The damping force provided by interaction with the water will result in an eventual return to the equilibrium state. In the case of a periodic exciting force, such as that found when a ship is moving through ocean waves, the vessel will not return to equilibrium but will instead reach a steady state condition in which it oscillates with a constant period and amplitude. This is referred to as forced, damped, heaving motion. It can be predicted mathematically by

$$z = Ae^{-vt}\sin\left(\omega_d t - \beta\right) + z_a\cos\left(\omega_e t - \varepsilon_2\right) \qquad \text{(Eq 7)}$$

where A is a constant determined from the initial conditions, v is a decay constant, ω_d is the circular frequency of the damped oscillation, b is the phase angle, z_a is the amplitude of the forced motion, ω_e is the circular frequency of the exciting force, and ε_2 is the phase angle of the forced motion relative to the exciting force [BHAT78]. In the steady state

condition the first term has decayed away and can therefore be ignored for the purposes of this model.

3. Pitching and Rolling

Pitching and rolling are also considered pure oscillatory motions. In these angular motions, notwithstanding, the moments rather than the forces must be considered. In both pitching and rolling, the angular motion results from the action of the following moments:

1. Inertial moment.

2. Damping moment.

3. Restoring moment.

4. Exciting moment.

The pitching moment is described in the equation

$$a\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + c\theta = M_0 \cos\omega_e t. \qquad \text{(Eq 8)}$$

The first term describes the inertial moment and is the product of the virtual mass moment of inertia a and the angular acceleration of pitching $\frac{d^2\theta}{dt^2}$. The second term describes the damping moment and is the product of the damping moment coefficient b and the angular velocity $\frac{d\theta}{dt}$. Likewise, the third term is the restoring moment in which c is the coefficient and q the angular displacement. These three terms will, in equilibrium, balance the exciting force which can be seen to vary in time with the encounter frequency ω_e.

The equation of motion for rolling is virtually identical to that for pitching, since it is a result of the same moments. It is important to note, however, that the coefficients for the various moments, a, b, c, and M_0 are determined separately for each kind of motion [NEWM77].

When a ship is travelling through waves, the motions of pitch and roll reach a steady state condition just as the heaving motion discussed earlier. When this steady state condition is reached, the equation for pitching becomes simply

$$\theta = \theta_a \sin\left(\omega_e t - \varepsilon_2\right) \qquad \text{(Eq 9)}$$

and rolling is defined by

$$\phi = \phi_a \sin\left(\omega_e t - \varepsilon_2\right). \text{ [BHAT78]} \qquad \text{(Eq 10)}$$

B. DETERMINING THE AMPLITUDE OF PERIODIC SHIP MOTION

As was previously discussed in the overview of marine vehicle dynamics, a vehicle's behavior when experiencing the effects of heave, pitch, and roll can during steady state conditions be described by simple periodic functions. The amplitudes by which these functions are multiplied, specifically z_a, q_a, and f_a, will vary with the angle of encounter with the seas and with the energy of the wave spectrum. It is the determination of these amplitudes, then, that is the key to realizing a workable model for approximating periodic ship motion in a computer application.

The values obtained for the heave, pitch and roll amplitudes are of keen interest to naval architects, since it is these values that determine the stability of ships in a seaway. Excessive degrees of roll can result in capsizing ships in rough seas, just as excessive pitch will result in shipping water over the forecastle, causing equipment damage and possible loss of life, or propulsion system degradation due to the screw coming out of the water.

There are two generally accepted methods for determining the heave, pitch, and roll coefficients, the analytical method (using strip theory) and data obtained from model tests. In either case the coefficients, referred to as response amplitude operators (RAO) or the transform spectrum, are fixed for a particular vessel, encounter frequency, and direction [BHAT78]. This leads to the conclusion that the RAO's, however determined, will be static for the duration of the computer simulation. An ideal method of implementing the RAO

27

values in the application, then, is by reading the values into an array from an external file during initialization.

C. MODELING THE EFFECT OF CONTROL SURFACES AND PROPULSORS ON SHIP MOTION

In addition to the periodic motions discussed in this chapter, the effects of control surfaces and propulsors must also be modeled in order to achieve realistic control over the virtual vehicle. It is a trivial matter to accept a user command to turn the ship or change the speed and implement this using a fixed rotation or translation rate, respectively. Unfortunately, this results in an instantaneous change in the speed or turning rate, and effectively negates the effect of immersion into the virtual world. The behaviors exhibited by the vehicle, while they do not have to be 100% scientifically accurate, must approximate real world behavior closely enough to be both believable and effective in training for the real world.

In the case of propulsors, the basic effect on the ship is a force along the longitudinal axis of the ship, either ahead or astern. The resulting motion has been previously described as surge. The magnitude of the force is a function of the blade speed and the blade pitch, as well as the initial velocity of the fluid through which the blades are turning. In gas turbine powered ships, the screw turns at a constant speed while the blade pitch is adjusted to change the thrust; conversely, a steam powered ship maintains a constant pitch and varies the shaft turning rate. As an illustration of how fluid velocity affects propulsor effect, consider a ship which is dead in the water suddenly increasing its turns to 200. The resulting energy will be to a large extent dissipated in cavitation and wave formation. On the other hand, a ship which is already making 10 knots through the water will convert more of this energy into useful thrust.

While the propulsor moves the ship over the ground, the rudder of a ship imposes a torque on the hull, resulting in an angular motion about the vertical z axis. This motion has previously been described in terms of yaw. Rudder design is predicated upon the ship

making headway. Rudder motion has an effect on ships making sternway, and even on ships that are virtually dead in the water, although the effects are less predictable, and vary with the currents and degree of backwash from the screw. Even in real life shiphandling, the effects of rudder motion on ships making sternway are difficult to predict, and vary widely according to ship design.

The torque imposed on the ship by the rudder varies with the speed of the ship and the angle of the rudder relative to the direction of motion. Torque is simply a translation of the energy imparted by the surrounding water molecules on the rudder surface, either through a pressure differential or by direct impingement.

For the purposes of a real-time simulator, a high degree of realism can be obtained for both propulsors and control surfaces by computing the forces and torques as a function of blade rate, ship's speed, rudder angle, etc. These values can then be summed with other forces and torques, mainly due to damping, to produce an acceleration for the non-periodic motions of the ship. The acceleration is then translated into a per-frame velocity and per-frame position by the Runge-Kutta method of solving second order ordinary differential equations (ODE) [PRES88]. The C++ implementation of the Runge-Kutta method is shown in Figure 10.

```
void Runge_Kutta_4th_order
              (float (*f)
                      (float,float, float),
              float (*g)
                      (float, float, float),
              float *x0,float t0,float *y0,
              float h)
{
// *f and *g are the two functions we want to
// integrate. h is the step size. Results are
// returned in *x and *y
      float k1 = f(*x0, t0, *y0);
      float l1 = g(*x0, t0, *y0);

      float k2 = f(*x0 + h * 0.5f * k1,
                      t0 + 0.5f * h,
                      *y0 + h * 0.5f * l1);
      float l2 = g(*x0 + h * 0.5f * k1,
                      t0 + 0.5f * h,
                      *y0 + h * 0.5f * l1);

      float k3 = f(*x0 + 0.5f * k2,
                      t0 + 0.5f * h,
                      *y0 + h * 0.5f * l2);
      float l3 = g(*x0 + 0.5f * k2,
                      t0 + 0.5f * h,
                      *y0 + h * 0.5f * l2);

      float k4 = f(*x0 + h * k3,
                      t0 + h, *y0 + h * l3);
      float l4 = g(*x0 + h * k3,
                      t0 + h, *y0 + h * l3);

      *x0 = *x0 + (h/6.0f) *
              (k1 + 2.0f*k2 + 2.0f*k3 + k4);
      *y0 = *y0 + (h/6.0f) *
              (l1 + 2.0f*l2 + 2.0f*l3 + l4);
}
```

Figure 10 C++ Implementation of 2nd Order ODE using the Runge-Kutta Method

D. USING THE WAVE OBJECT FOR VEHICLE MOTION

As was described in Chapter III, the wave form is implemented as an object separate from the implementation of the ocean surface itself. This decision was made since the periodic motions of the marine vehicle will, in steady state, have a period equal to that of the exciting force. In this case, of course, the exciting force is the wave form. If it is extended to take into account the encounter frequency the ship sees as it travels through the waves, and also takes into account the angle of incidence to the waves, the wave object can return not only the wave height at a particular point on the surface, but also the magnitude of heave, pitch, and roll motions of the ship. Any changes to the wave object parameters, such as amplitude, wavelength, and direction will be reflected equally in the ship motions and the ocean motion.

V. The OSF/MOTIF USER INTERFACE

A. OVERVIEW OF MOTIF

Motif is a graphical user interface (GUI) produced by the Open Software Foundation, a non-profit organization founded in 1988 by such industry giants as IBM, Digital Equipment, and Hewlett-Packard. Built on top of the X Window System (a network-based window system developed at MIT), Motif uses a set of user interface components referred to as widgets to provide the framework of the GUI, and allow the developer to use a set of cross-platform tools and standards to simplify the development of GUI-based applications. [McMI92]

A widget, in the Motif sense, is similar to a C++ class, in that it consists of both data structures and procedures. While most widgets are visible in the form of a window, some widgets are superclass widgets and serve only to provide resources and other characteristics for subclasses to inherit [McMI92]. Visible widgets take on such familiar GUI component forms as windows, scroll bars, pushbuttons, toggle buttons, dialog boxes, sliders, and menu bars. Motif provides a truly rich, and more importantly, standardized set of these widgets that software developers can use to create an application whose interface is already familiar to the X workstation user.

Callbacks provide the functionality in a Motif application. A callback is simply a user defined function which is called by the application if a particular event occurs within a widget, such as the pressing of a push button or the sliding of a scale. When the callback is called, it is also supplied with two structures, one of which contains information about the event which caused it to be invoked, and one which contains any user-defined data the callback might need [YOUN92].

If a Motif application should perform processing continuously in the absence of input events, such as in an animation program, the user can supply the appropriate function as a work procedure. The work procedure, defined and supported through the Xt Intrinsics, will be invoked whenever the X event queue is empty. Another type of user-supplied function,

32

referred to as a timeout, will be invoked after a specified number of elapsed clock ticks, regardless of the condition of the X event queue [HELL91].

B. A MOTIF APPLICATION FRAMEWORK LIBRARY

While Motif provides a wealth of user interface components that allow the application writer to create a standard GUI for his software product, a typical C/C++ application program written with on the Motif library involves a lot of repetition. For instance, to create a simple one-line text widget, which is positioned at certain location in a bulletin board widget, and has a specific number of columns, the code would look something like the lines seen in Figure 11. The process of creating a widget set, registering callbacks, setting widget resources, and realizing widgets is fairly well defined, and as such should lend itself to being encapsulated in the form of reusable code. Douglas Young, a member of the technical staff at Silicon Graphics, Inc. has taken this concept and created an entire application framework library to simplify the creation of Motif applications [YOUN92]. Taking an object-oriented approach to define the general architecture of a Motif application, Young has created such base classes as UIComponent to provide common functionality to any widget component, such as managing and unmanaging, creation and destruction, and resource allocation. Two major classes derived from UIComponent are the Application class and the MainWindow class. Again, these higher level classes seem fairly simple but provide the framework for many different types of applications. The Application class allows the registering of multiple MainWindow objects, so that a single application can consist of more than one main window. The application framework library is available by anonymous ftp, and a detailed description of its individual classes can be found in Young's *Object-Oriented Programming with C++ and OSF/Motif* [YOUN92].

```
int n=0;

XtSetArg(args[n], XmNy, 150); n++;

XtSetArg(args[n], XmNcolumns, 40); n++;

XtSetArg(args[n], XmNrows, 1); n++;

text = XmCreateText(bboard, "text", args, n);

XtManageChild(text);
```

Figure 11 Creating a Text Field Widget in C

C. A COMPARISON OF PERFORMER AND MOTIF APPLICATIONS

A good beginning for comparing two application types is in the control structure. A Motif application, no matter how small or large, consists of four phases:

1. Initializes Xt Intrinsics.

2. Creates widgets and registers any callbacks.

3. Realizes the widgets.

4. Processes events in the application's main loop.

A Performer application also has a fairly rigid program structure:

1. Performer initialization and configuration.

2. Creates a visual database consisting of pfNodes and registers any callbacks.

3. Executes the Performer main simulation loop, in which the frames are computed, culled, and drawn.

In many ways a Performer application closely parallels a Motif application, particularly when Performer runs as a single-threaded process.

A second criterion for comparing Motif and Performer is the object-oriented paradigm. Although the Motif and Performer libraries are written so they can be included

in both C and C++ applications, and therefore are not presented as C++ classes, they still demonstrate an adherence to the concept of inheritance that has proven so useful in recent years. Motif widgets are linked by a well defined class hierarchy, including a few abstract classes that cannot be directly instantiated, such as the Composite, Core, and Constraint widgets. These higher level widgets, however, define behaviors and resources that are inherited from lower level widgets [McMI92]. In a similar fashion, Performer pfNodes form the top level of the Performer node hierarchy and cannot be directly instantiated. Rather, the pfNode provides a repository of properties inherited by all nodes, such as name, intersection mask, bounding geometry, callback functions, and callback data [SGI92a].

D. PROCESS CONTROL CONSIDERATIONS

As was stated earlier, Performer and Motif applications are quite similar in form when a Performer application is running single-threaded. In a typical Motif application, the program is designed for sequential execution. This is evidenced by the structure of the application main loop. Within the main loop, the application simply checks the X event server queue for events that need to be processed and invokes the associated callback functions in the order in which they appear on the widget's callback list [HELL91].

In a Performer application, however, the default mode is to fork separate application, cull, and draw processes. The control over program execution rests in the application process, where the call to pfSync in the Performer main loop causes each process to sleep (if necessary) until the parallel threads have finished processing their respective frames [SGI92].

E. MAJOR ISSUES TO BE RESOLVED

Since the Motif and Performer applications have such disparate notions of process control in the application main loop, a paradigm that permits both levels of control must be found before the two can be successfully integrated. The issue of graphics control is also central to the goal of integrating Motif and Performer in a single application, since it also complicates the handling of user input. Chapter VI will explore these issues further and

provide the details of how they can be resolved to successfully implement Performer with a Motif user interface.

VI. IMPLEMENTING PERFORMER IN AN X WINDOWS APPLICATION USING MOTIF

A. CUSTOMIZATION WITH COMMAND LINE OPTIONS

Motif applications have a built-in ability, thanks to the Xt Intrinsics, to allow the user to set certain "look and feel" parameters such as background color and border width in the command line at execution [McMI92]. In addition to the standard command line options supplied by the Xt Intrinsics, the programmer can specify additional resources for the command line within the program code. This proves particularly beneficial when running a Performer application within Motif, since start-up parameters such as multiprocessing mode, number of pipes, window size, data file names, etc. can be set up to be accepted as command line input. To accomplish this command line setup, the programmer must follow some fairly straightforward guidelines that can be found in [NYE90]. The data structures and code fragments illustrating this technique for setting the Performer multiprocessing mode are shown in Figure 12.

Command line options, of course, are not the only way to specify Xt resources. They can also be supplied in application specific app-defaults files, user specific .Xdefaults files, or hard coded into the program with XtSetArgs or XtSetValues function calls. This fact can also be exploited in a Performer application, locating default values for Performer initialization and configuration in the app-defaults file.

B. THE GLXCONFIG STRUCTURE AND GRAPHICS INITIALIZATION

When the Performer graphics pipeline is initialized, a callback function, previously registered with the pfInitPipe call, is used to open a GL window and configure it appropriately. If the *initfunc* argument to pfInitpipe is NULL, Performer will open the full screen configured by the provided function pfInitGfx [SGI92]. The *initfunc* is registered as

37

```
typedef struct {
    int        mp_mode;
} AppData, *AppDataPtr;


static XtResource resources[] = {
("mp_mode", "Mp_mode", XtRInt, sizeof(int),
  XtOffset(AppDataPtr, mp_mode), XtRImmediate, (caddr_t)-1 },
};


static XrmOptionDescRec options[] = {
{ "-mp", "*mp_mode", XrmoptionSepArg, NULL ),
};


 void
 main(int argc, char ** argv)
 {
        AppData appData;
 Widget toplevel;
 toplevel = XtVaAppInitialize(& app_context,
            "PfApp",
            options,
            XtNumber(options),
            &argc, argv,
            NULL);

 XtGetApplicationResources(toplevel, &appData,
            resources, XtNumber(resources),
            NULL, 0);

 if (appData.mp_mode >= 0)
        pfMultiprocess(appData.mp_mode);
 pfConfig();
```

Figure 12 Setting Command Line Options for a Performer Application

a callback, so that it may be called from the separate draw process, since the draw process
must have exclusive ownership of its GL context [OLSE93].

If Performer is to be integrated into a Motif application, it has to be configured to render into a widget. To allow Silicon Graphics GL applications to run in a Motif application, a special widget called GlxMdraw is provided. It provides a window with the appropriate visual and colormaps needed for GL, based on supplied parameters. GlxMdraw also provides callbacks for redraw, resize, input, and initialization. GlxMdraw is a subclass of the Motif widget class XmPrimitive and has resources and defaults suitable for use with Motif, such as the default Motif background and foreground colors [SGI93]. The GlxMdraw widget is one element of what Silicon Graphics refers to as GLX mixed model programming.

Rather than use the provided function pfInitGfx to handle the GL configuration, a structure known as GLXConfig is supplied as a resource when instantiating the GlxMdraw widget. The GLXConfig structure contains the configuration information needed to create and render GL into an X window [SGI93]. This structure will be initialized with the values which, when the widget is realized, will result in the configuration needed by Performer.

C. SHIFTING GRAPHICS PIPELINE CONTROL BETWEEN PROCESSES

As was mentioned at the close of Chapter V, a key issue in achieving the goal of integrating Motif and Performer is the control over the graphics process. The draw process must have exclusive hold on the graphics pipeline in order to perform its function [OLSE93], therefore the GL window is normally created in the user-supplied initialization function when it is invoked as a callback by the draw process. Input events are handled by adding input devices to the GL device queue, and polling this queue during the draw callback. This arrangement is less than ideal, since input events are normally meant for handling by the application process. NPSNET handles this issue by creating its own event queue from lockable shared memory (i.e. a pfDataPool), entering into this new queue any events that must be handled by the application process [YOUN93].

A Motif application, on the other hand, is written so that input events are handled in the application process, specifically in the XtAppMainLoop. A way must be found, then, for the Performer draw process to handle graphics calls while the application process handles input events. To achieve this result, two function calls are used, the GLXlink and GLXunlink functions. These functions allow processes to attach to and detach from the GL graphics context. Using these functions, one creates the Motif widget set within the application process, and then switches control of the GlxMdraw graphics context from the application process to the draw process when the widget is realized. Since the first event associated with the realization of the GlxMdraw widget is the ginit event, a callback registered for that event is a convenient place to perform this context switch.

The only data necessary to transfer between processes, via shared memory, is a simple structure that contains information about the created GlxMdraw widget sufficient to provide the necessary arguments to the GLXlink function. As shown in Figure 13, the ginit callback function obtains this data from the created widget and places it in the shared memory structure. It calls the GLXunlink function, indicating to the system that the process never intends to perform GL drawing in the window again [SGI93]. Following this call, the pfInitPipe function is called, registering a user-supplied function as the *initfunc* which in turn will take the information from the shared data to call GLXlink. Calling GLXlink will tell the system to allocate to the calling process those resources necessary for GL rendering [SGI93]. The final result is a GlxMdraw widget, which is created and managed in the application process, but whose graphics context and associated system resources are allocated to the draw process.

D. EXTENDING THE MOTIF APPLICATION FRAMEWORK

One of the goals of this research was to extend the object-oriented paradigm of NPSNET beyond its present limits of entity behaviors and user interface device drivers [YOUN93]. Program architecture can also be controlled in an object-oriented paradigm through the use of virtual functions. By defining virtual functions in an abstract class,

```
// This member function is called at widget realization
// it holds the key to switching control of the GL context
// to the draw process
void PfWindow::init(Widget w, GlxDrawCallbackStruct * cb)
{
 *win_x_size = cb->width;
 *win_y_size = cb->height;


 // Disconnect from the GL widget and allow the draw
 // process to connect to it
    Display * display = XtDisplay(w);
    Window xWindow = XtWindow(w);


// Place info in shared memory so draw process can attach
// to GLXwidget.
    glx_info->display_name = XDisplayName(NULL);
    glx_info->xWindow = xWindow;


// Release exclusive hold on GLXwidget.
    GLXunlink(display, xWindow);


// Performer will now call openGLXconnection in the
// draw process.
    pfInitPipe(pipe, &PfWindow::OpenGlxConnection);
```

Figure 13 Detaching from the GL context in the Application Process

program structure can be specified in a high-level manner. If the function is merely virtual, a default form of the function can be specified that might serve as a "working copy." Derived classes can use the function "as-is," or choose to redefine the function to accomplish a different set of tasks. Derived classes can also extend the function by calling the base class' function within the redefined function, followed by additional lines of code.

41

```
// Called by the draw process. Allows the draw process
// to get the necessary information about the GlxMDraw
// widget and attach to the GL context.
void PfWindow::openGlxConnection(pfPipe * p){
 Display *display = XOpenDisplay(glx_info->display_name);
 Window glx_window = glx_info->xWindow;
 XWindowAttributes attributes;
 XGetWindowAttributes(display, glx_window, & attributes);
 int screenNo = XScreenNumberOfScreen(attributes.screen);


// Use the same configuration here that was used
// in creating the widget.
 GLXconfig * config;
 config = GLXgetconfig(display,
                          screenNo, regularGlxConfig);
// Find the window entry and set it to have the same
// window id of the GLXwidget's window.
 for (int i = 0; config[i].buffer; i++)
    if (config[i].buffer == GLX_NORMAL &&
                       config[i].mode == GLX_WINDOW)
        config[i].arg = (int)glx_window;
// Connect the GL context to the GLXwidget
 (GLXlink(display, config);
// Make it the current window.
 GLXwinset(display, glx_window);
 zbuffer(TRUE);
}
```

Figure 14 Attaching to the GL context in the Draw Process

The program structure remains fixed by the base class; polymorphism allows derived classes to follow the same basic structure with more specialized results.

1. The PfApplication class

The result of this research was an extension to the MotifApp library defined by Young [YOUN92]. The Application class was used as a base class from which was derived the PfApplication class. As an unusual feature, the MotifApp library already includes the main() function, which presumes the existence of a global pointer to an instance of the Application class, referred to as *theApplication*. The main() function simply calls the initialize() and handleEvents() methods of *theApplication*. In terms of a Performer application, then, all the initialization and configuration, as well as database creation must occur in the initialize() method

```
void PfApplication::initialize (int * argcp,
                                char ** argv)
{
    int i; //loop variable

    pfInit();

    allocateSharedData();

    // Need for registered windows to allocate
    // their shared data also
    // before forking any cull and draw processes
    _pfWindows = (PfWindow **)_windows;

    for (i = 0; i < _numWindows; i++)
        _pfWindows[i]->allocateSharedData();

    pfMultiprocess(PFMP_DEFAULT);
    pfConfig();

    _scene = pfNewScene();
    Application::initialize(argcp, argv);

    postConfigSetup();

    addWorkProcs();

}
```

Figure 15 The PfApplication initialize() method

43

a. allocateSharedData

Most Performer applications will need to allocate shared data for the purpose in inter-process communication. While this step is not absolutely necessary to have a successful application, it *is* vital that the allocation occur after the pfInit call and prior to the forking of individual processes at pfConfig. The empty virtual function allocateSharedData provides the venue for this task. It is defined and left blank in the base class so that a derived class need only implement this step if it needs to allocate specific shared data structures.

b. postConfigSetup

The next virtual function encountered in the initialize method is the postConfigSetup function. Again, this particular area of a Performer application is wide open to the details of the particular program objective. The only restriction is that this function contain the Performer calls that must follow the pfConfig call (hence the name) and are performed prior to the simulation main loop.

c. addWorkProcs

Work procedures are the key to resolving the issue of process control between Performer and Motif. As stated in Chapter V, a work procedure is a type of callback that is invoked in the X event loop whenever there are no events to be processed. So the Performer main loop can be implemented as the body of a work procedure which is never removed. As long as the X event queue can be emptied, the work procedure can be called often enough to sustain the desired frame rate for the Performer application. During testing, even a continuously depressed key did not cause any slowdowns in frame rate.

d. preSimulationSetup

As stated above, certain Performer calls must occur after the call to pfConfig and prior to the simulation main loop. The postConfigSetup function was provided to allow the derived class to provide the specific details of this phase. In the case of NPSNET, this phase takes seventy two seconds, performing pre-loading of files to minimize response

time during program execution. But such a lengthy delay before the opening of the windows is undesirable, since it may lead the user to think the system has "locked up." The preSimulationSetup function is provided to alleviate this problem.

Called by the postWindowSetup work procedure, which removes itself upon completion, the preSimulationSetup function will be called after the widget set is realized, and before the PerformerMainLoop work procedure. Any calls that can be made in the postConfigSetup function can be made in preSimulationSetup as well, except they will now occur after the widgets are realized.

e. LoadDatabase

The loadDatabase is a pure virtual function, causing the PfApplication class to be an abstract class. It should be intuitively obvious that there is no such thing as a generic simulation. You have to simulate *something*, so the programmer must supply the body for the loadDatabase function in his derived class.

It is an arbitrary decision whether the loadDatabase function is called within postConfigSetup or postWindowSetup. In the PfApplication class, loadDatabase is called by postWindowSetup in order to avoid any unwanted delays in the opening of the Performer display. Like any other virtual function in the class, however, this can be overridden in the derived class.

f. PerformerMainLoop

The main loop of a Performer application must call pfFrame to cause the next frame to be rendered. If pfSync is called separately from pfFrame, as shown in Figure 16, any latency-critical processing must occur after pfSync but before pfFrame [SGI92a]. The empty virtual function doLatencyCriticalUpdates is provided to allow derived classes to perform this processing, if it is so desired. The remainder of the per-frame processing (i.e. updating vehicle dynamics, time of day, etc.) is accomplished in the empty virtual function doSimulation.

```
Boolean PfApplication::PerformerMainLoop()
{
    doLatencyCriticalUpdates();
    pfFrame();
    doSimulation();
    pfSync();
    return FALSE;  // prevents work procedure from
                   // being removed
}
```

Figure 16 The Performer Main Loop

2. Handling callbacks

The PerformerMainLoop function, and for that matter all functions implemented as callbacks in Performer and Motif, presents a minor difficulty when using C++ classes. C++ member functions have a hidden first argument, whose purpose is to hold the *this* pointer to the instance of the class. If the member function is called from C, as it is when invoked as a callback, the *this* pointer is not supplied, causing the remaining arguments to be incorrect [YOUN92]. This turns out to be the case in both Performer and Xt callbacks.

Member functions *can* be used to define the behavior resulting from a callback, but at the added overhead of another function call. The static functions _PerformerMainLoop and _postSimulationSetup provide this feature by accepting the *this* pointer as client data, and using the pointer to call the desired member function. This method of handling callbacks within C++ classes is used exclusively in the MotifApp library [YOUN92], and can also be used for Performer callbacks when client data can be supplied.

3. The PfWindow Class

In addition to the creation of the GlxMDraw widget discussed above, the PfWindow class provides a static wrapper function as callback for the GlxMDraw's input event, which in turn calls an empty virtual function *input* which the derived class can redefine for itself. Most of the functionality of the PfWindow class, in fact, is defined in the

46

higher level MenuWindow and MainWindow classes, for it is in these classes that the component Motif widgets are handled [YOUN92].

VII. NPSNET IN A MOTIF APPLICATION

After some initial testing with the extended MotifApp library on a simple Performer application, the results were extremely satisfying. The question remaining was whether NPSNET could be "reverse-engineered" to fit into this framework, and if so, what changes needed to be made. This chapter summarizes those changes.

A. CHANGES TO THE MAIN PROCEDURE

In the MotifApp library the main() function is already defined. Therefore, the NPSNET main() function had to be altered so that its behavior could be desrcibed in terms of the Application class methods initialize() and handleEvents().

The first step in this transformation was the creation of the NPSNETApplication class, a derived class from PfApplication. Variables that were once local to the main() function were transformed into members of this class. Then the statements within the main() function had to be parsed into the sections preceding the call to pfConfig, those statements after pfConfig up to the time the window is open for drawing, and the statements from window opening to the beginning of the main loop. Pipe and channel configuration was made a part of the NPSNETWindow class initialization method.

The NPSNETApp class does not need to fully utilize the level of abstraction provided by the PfApplication class. Since the loading of database files, texture information, etc. is already well-defined and integrated with the introduction frames of NPSNET, there is no need to use the loadDatabase function. Since this is a pure virtual function, however, and *must* be defined, it is defined as an empty function. The function is never called anyway, since the NPSNETApp class redefines the postWindowSetup function.

The body of the original NPSNET main simulation loop is separated into the latency-critical and application-specific portions, and allocated to the bodies of doLatencyCriticalUpdates and doSimulation respectively. The layer of abstraction

provided by the base class PfApplication is used quite easily, and enforces the Performer application architecture to a greater extent.

Finally, the exit code executed after NPSNET receives the signal to exit is taken out of the class, and placed in the body of the doit() function of the NPSNETQuitCmd class. The NPSNETQuitCmd is a derived class from the MotifApp library's QuitCmd class. It inherits all the behaviors of the parent class, but adds the code necessary to cleanly terminate all forked processes and call pfExit.

B. REVISING THE USER INPUT PARADIGM

NPSNET-IV was designed to run in a GL window, with user input handled by the GL device queue. As the draw process owns all resources associated with GL rendering, the user input occurs in the draw process. Because of this arrangement, a user-defined device queue allocated from shared memory was created to allow inter-process communication about user input to take place between the application and draw processes [YOUN93].

In the Motif version of NPSNET, user input is now handled by the X event queue in the application process, and the callbacks registered with these events provide the handling of the input. Individual commands such as weapon firing, relocating, and environment control can be taken out of the NPSNET main loop and distributed to objects derived from the Cmd class. Not only does this simplify the main loop, it allows multiple user interface components, such as hot keys, menu buttons, and scale widgets to operate on the same command object. Fewer global state components are necessary in this arrangement, since each command object can maintain state information about itself.

C. THE NPSNET WINDOW

Keyboard and mouse input is handled by the GlxMDraw widget with its input callback function, so one of the chief functions of the NPSNETWindow class during execution is the handling of input events.

Prior to the simulation loop, though, the graphics pipeline and associated channels must also be configured. Since the pfPipe is associated with one particular window rather

than the main application, it follows that pipe and channel calls be handled by the NPSNETWindow. As a derived class of the PfWindow, more than one instance of the NPSNETWindow class can be registered with the NPSNETApplication. This would be desirable particularly if the workstation has multiple hardware graphics channels. Currently the NPS Graphics and Video Laboratory has one Silicon Graphics Onyx Reality Engine with a multichannel option, and research is ongoing as to how best to utilize this feature. Meanwhile, the PfWindow class and its derived NPSNETWindow class will operate with a single pipe only.

VIII. CONCLUSIONS

A. RUN-TIME PERFORMANCE

The advantages of using a GUI versus a keystroke interface in NPSNET is apparent once you begin to use it. But it must not come at the expense of degraded simulation performance. To discover whether there were any performance advantages or penalties associated with the revised version of NPSNET, the old and new versions were run on the same machine. The Performer DrawChannelStats function was used to display performance data at four stationary locations in the NPSNET world, the airport, canyon, village, and pier. The values were then compared, both with and without the addition of the ocean object. Table 1 clearly shows that a Motif interface imposes no significant change in the performance of NPSNET.

Table 1: PERFORMANCE STATISTICS TAKEN AT FOUR STATIONARY LOCATIONS IN NPSNET

NPSNET Performance Statistics		
	NPSNET IV.2	NPSNET IV under Motif
Airport	15.0 hz sustained, 10.0 hz min	15.0 hz sustained, 10.0 hz min
Canyon	15.0 hz sustained, 8.6 hz min	15.0 hz sustained, 7.5 hz min
Village	10.0 hz sustained, 5.5 hz min	10.0 hz sustained, 6.7 hz min
Pier	20.0 hz sustained, 10.0 hz min	20.0 hz sustained, 12.0 hz min

B. LIMITATIONS

1. Wave complexity

As the number of wave components is increased, the ocean surface takes on more of a random, confused appearance, thus adding to the realism of the scene. However, this added realism comes at the expense of multiplying the total animation time by the number of wave components specified. With an ocean modeled as an array of 1296 vertices and the

wave form created as a single component, the NPSNET application process can process a single frame in 0.03 to 0.05 seconds, allowing a frame rate of twenty to thirty frames per second. If the same ocean is animated with a wave form of seven components, however, the application time per frame increases to approximately 0.14 seconds, allowing a frame rate of only seven frames per second. The level of detail of the ocean in terms of number of vertices, and the realism of the wave form in terms of number of components, must be chosen to achieve an acceptable compromise between frame rate and realism.

It should be pointed out here that the ocean database is organized both as a spatial hierarchy and a simple array of vectors. Since each point in the array is computed as a periodic function independent of other points, and because the vector array resides in shared memory, the task of updating wave heights can be parallelized, reducing the computation time significantly. As the number of processors available on a workstation increases, therefore, the realism of the ocean can be increased accordingly.

2. Callbacks and C++

As discussed in Chapter VI, callbacks registered with X and Performer cannot be C++ class member functions. A static member function was used with the *this* pointer given as the client data argument, so that a member function could be called "second-hand." This arrangement is inefficient but necessary under the current versions of X and Performer.

Since user input events are infrequent, there is generally no significant performance penalty. However, this could pose a significant problem in Performer, where pre-cull, post-cull, pre-draw, and post-draw callbacks can be registered. This can build up to a significant performance penalty, depending on the number of nodes with callbacks, and which process is most limiting for the desired frame rate. For instance, if an application's frame rate is limited by the cull process, the additional overhead imposed by an extra function call during each pre-cull callback could cause the simulation to pass a frame boundary and skip frames, seriously degrading performance. The draw process, on the other hand, is not generally CPU intensive and the only callbacks executed are for those

nodes that are added to the display list by the cull process. The extra overhead, then, is not so significant.

One area where the static function wrapper is inadequate when implementing Performer in C++ classes is the *initfunc* supplied for the pfInitPipe function call. In this case there is no provision for supplying client data as a parameter to the *initfunc*. A workaround was used in this implementation where a global pointer was initialized in the pfWindow's constructor to point to the instantiated object. This imposes a limitation of a single instance of the pfWindow class to which Performer can draw. Since most of the workstations at the NPS Graphics and Video laboratory have a single hardware pipeline, this limitation does not hinder existing projects. A future workaround might be a dynamically allocated array of global pointers and process ID's (PID) of size *numPipes*, where *numPipes* is also the argument supplied to pfMultipipe. This array could then be searched in the static function for a match of its own PID, and use the corresponding instance pointer to call the correct member function.

3. System compatibilty

Under IRIX 4.0.5G, calls to the Iris Font Manager library caused a segmentation fault under the GLX mixed model paradigm. The problem did not occur under IRIX 5.1.1, which is currently installed on the two Onyx Reality Engine workstations in the lab. Consequently, the Font Manager library is not utilized under the 4.0.5 operating system. Silicon Graphics has been informed of this problem, and future operating system upgrades should alleviate the situation.

C. SUGGESTIONS FOR COMMERCIAL SOFTWARE UPGRADES

As the advantages of object-oriented programming allow C++ to gain in popularity among software developers, Motif and the supporting Xt Intrinsics need to provide a means of registering C++ class member functions as callbacks, and avoid the redundancy and overhead involved with the present method of using a static function as intermediary. The same is true with the Performer library.

An obvious alternative is representing Motif widgets and Performer nodes as classes themselves, with the callback represented as a member object which is registered with the class. This relationship is analogous to that between the PfApplication and PfWindow classes in the application frameworks library.

D. FUTURE WORK

This work provides a model for a more realistic ocean surface and corresponding ship motion, built on a framework that is highly structured and extensible. Some of the areas for future work are:

- Parallelizing the Ocean class animate() method to allow multiple wave components without peformance penalty.

- Converting the existing NPSNET input structure into a library of command objects, and creating new Motif-based user interface components to execute them. This has been only partially implemented, as a proof-of-concept.

- Extending the wave model to incorporate water depth to better approximate wave behavior near shorelines.

- Developing a means of loading, creating and saving configuration information interactively for marine vehicles, such as RAO data, moments of inertia, shaft horsepower, etc.

- Develop simulated weapons and sensors for ships and submarines for use in the NPSNET environment. This may also invlove the creating of Motif-based widgets to handle input and displays.

This work continues the evolution of NPSNET, and lays the foundation for faster prototyping of any Performer-based application. As hardware and software capabilites expand, the unlimited imaginations of future researchers will result in a distributed simulation system that continues to find new uses in a modern military.

APPENDIX A. THE PFAPPLICATION CLASS SOURCE CODE

```
/********************************/
/*This is file PfApplication.h   */
/********************************/
2 #ifndef _PFAPPLICATION_H
3 #define _PFAPPLICATION_H
4
5 #include "Application.h"
6 #include <Performer/pf.h>
7
8 class PfWindow;
9
10class PfApplication : public Application
11{
12
13    friend class PfWindow;
14
15#if (XlibSpecificationRelease>=5)
16    friend void main ( int, char ** );
17#else
18    friend void main ( unsigned int, char ** );
19#endif
20
21
22
23    protected:
24
25       virtual void allocateSharedData() {};
26       virtual void loadDatabase() = 0;
27       virtual void postConfigSetup() {};
28       virtual void preSimulationSetup() {};
29       virtual void doLatencyCriticalUpdates() {};
30       virtual void doSimulation() {};
31
32       static Boolean _PerformerMainLoop(XtPointer);
33       static Boolean _postWindowSetup(XtPointer);
34
35       static void mapCB(Widget, void *, XEvent *);
36       virtual void map(XEvent *);
37
38
39
40       virtual Boolean PerformerMainLoop();
41       virtual Boolean postWindowSetup();
42
43       PfWindow ** _pfWindows;
44
45       void addWorkProcs();
```

```
46
47      XtWorkProcId workProc, tempWorkProc;
48
49      pfScene *_scene;
50    // Functions to handle Xt interface
51
52#if (XlibSpecificationRelease>=5)
53    virtual void initialize ( int *, char ** );
54#else
55    virtual void initialize ( unsigned int *, char ** );
56#endif
57
58      pfScene * scene() {return _scene;}
59
60    public:
61
62      PfApplication( char *);
63
64      virtual ~PfApplication(){};
65
66};
67
68extern PfApplication * thePfApplication;
69
70#endif
```

```
/**********************************/
/*This is file PfApplication.C    */
/**********************************/
1#include "PfApplication.h"
2 #include "PfWindow.h"
3
4 #include <Performer/pf.h>
5 #include <signal.h>
6 #include <stream.h>
7
8 extern PfApplication * thePfApplication = NULL;
9
10 PfApplication::PfApplication(char * name) :
11                                  Application(name)
12 {
13     thePfApplication = this;
14 }
15
16 #if (XlibSpecificationRelease>=5)
17     void PfApplication::initialize ( int * argcp,
18                                      char ** argv)
19 #else
20     void PfApplication::initialize (unsigned int * argcp,
21                                     char ** argv)
22 #endif
23 {
24     int i; //loop variable
25
26     pfInit();
27
28 #ifdef DEBUG
29     pfNotifyLevel(PFNFY_DEBUG);
30 #else
31     pfNotifyLevel(PFNFY_WARN);
32 #endif
33
34     allocateSharedData();
35
36     // Need for registered windows to allocate
37     // their shared data also
38     // before forking any cull and draw processes
39     _pfWindows = (PfWindow **)_windows;
40
41     for ( i = 0; i < _numWindows; i++ )
42       _pfWindows[i]->allocateSharedData();
43
44     pfMultiprocess(PFMP_DEFAULT);
45     pfConfig();
46
47     _scene = pfNewScene();
48     Application::initialize(argcp, argv);
```

57

```
49
50     XtAddEventHandler(baseWidget(),
51                   StructureNotifyMask, FALSE,
52                   (XtEventHandler)&PfApplication::mapCB, 0);
53
54     postConfigSetup();
55
56     addWorkProcs();
57
58 }
59
60 // This Xt callback is called when the application shell
61 // widget is stowed or unstowed.
62 void PfApplication::mapCB(Widget, void * userdata,
63                                    XEvent * event)
64 {
65     PfApplication * obj = (PfApplication *) userdata;
66
67     obj->map(event);
68 }
69
70 void PfApplication::map(XEvent * event)
71 {
72
73     static pid_t draw_pid;// draw process ID.
74
75                         // Xt calls draw_workproc()
76                         // when no events are pending.
77     if (event->type == MapNotify)
78     {
79                         // Resume the stopped draw process
80                         // (see below).
81       if (draw_pid > 0)
82           kill(draw_pid, SIGCONT);
83
84                         // Get the draw processes ID.  If
85                         // there is no draw process, don't
86                         // worry about suspending it.
87       else if (draw_pid == 0)
88       {
89           long mode = pfGetMultiprocess();
90
91           if (mode & PFMP_FORK_DRAW)
92             draw_pid = pfGetPID(0, PFPROC_DRAW);
93           else
94             draw_pid = -1;
95       }
96     }
97
```

```
98                        // Nothing is done when the
99                        // application is iconified.
100    if (event->type == UnmapNotify)
101    {
102                        // If there is a separate draw
103                        // process, it must be temporarily
104                        // stopped, otherwise it busy-waits.
105     if (draw_pid > 0)
106         kill(draw_pid, SIGSTOP);
107    }
108}
109
110void PfApplication::addWorkProcs()
111{
112
113    // This is how we implement the Performer main loop
114     // in a Motif Application.
115    workProc = XtAppAddWorkProc(_appContext,
116         (XtWorkProc)&PfApplication::_PerformerMainLoop,
117         (XtPointer)this);
118
119    // This takes care of one-time-only stuff after the
120     // application enters the XtAppMainLoop. It is
121     // added last so it will be executed first.
122    tempWorkProc = XtAppAddWorkProc(_appContext,
123         (XtWorkProc)&PfApplication::_postWindowSetup,
124         (XtPointer)this);
125
126}
127
128// These are the Xt work procedures that are called
129// after the Motif widgets are realized and the Xt event
130// loop begins. They are implemented as static
131// "wrappers" which receive a pointer to the object
132// instance and call the instance-specific methods.
133// This allows these methods to be redefined by
134// derived classes.
135
136Boolean PfApplication::_PerformerMainLoop
137                            (XtPointer clientdata)
138{
139    PfApplication *obj = (PfApplication *)clientdata;
140     return obj->PerformerMainLoop();
141}
142
```

```
143 Boolean PfApplication::PerformerMainLoop()
144 {
145     doLatencyCriticalUpdates();
146     pfFrame();
147     doSimulation();
148     pfSync();
149     return FALSE;   // prevents work procedure from
150                     // being removed
151 }
152
153 Boolean PfApplication::_postWindowSetup
154                                 (XtPointer clientdata)
155 {
156     PfApplication *obj = (PfApplication *)clientdata;
157      return obj->postWindowSetup();
158 }
159
160
161
162 Boolean PfApplication::postWindowSetup()
163 {
164     loadDatabase();
165
166     preSimulationSetup();
167
168     return TRUE;
169 }
```

APPENDIX B. THE PFWINDOW CLASS SOURCE CODE

```
/**********************************************************/
/*This is file PfWindow.h                              */
/**********************************************************/
1    #ifndef _PFWINDOW_H
2    #define _PFWINDOW_H
3
4    #include "MenuWindow.h"
5    #include <Performer/pf.h>
6    #include <Xm/Xm.h>
7    #include <X11/Xirisw/GlxMDraw.h>
8
9    class PfWindow : public MenuWindow
10   {
11           friend class PfApplication;
12     protected:
13           virtual void initialize();
14           virtual void createMenuPanes();
15           virtual Widget createWorkArea(Widget);
16
17           virtual void createCommandWindow();
18           virtual void setupChannel();
19
20           static void redrawCB (Widget,XtPointer,XtPointer);
21           static void inputCB  (Widget,XtPointer,XtPointer);
22           static void DrawChannel(pfChannel *, void *);
23           static void OpenGlxConnection(pfPipe *);
24
25           virtual void openGlxConnection(pfPipe *);
26           virtual void init(Widget, GlxDrawCallbackStruct *);
27           virtual void resize(Widget,GlxDrawCallbackStruct*);
```

```
28          virtual void input (XtPointer) {/* Empty */};
29

30          Widget      parent,
31                      _commandWindow,
32                      glw;
33          pfPipe *pipe;
34          pfChannel *channel;
35

36          static pfLight * Sun;
37          static pfEarthSky * ESky;
38          float       chanNear, chanFar,
39                      chanFOVhorizontal,
40                      chanFOVvertical;
41          void  *     arena;
42   public:
43          PfWindow(char * name);
44          pfChannel * theChannel() {return channel;}
45          virtual void allocateSharedData();
46   };
47

48   extern PfWindow * thePfWindow;
49

50   typedef struct {
51    char * display_name;
52    Window xWindow;
53   } glx_info_struct;
54

55   extern glx_info_struct * glx_info;
56

57   extern int * win_x_size;
58   extern int * win_y_size;
59   #endif
```

```
/***********************************************************/
/* This is file PfWindow.C                                 */
/***********************************************************/
1   #include "PfWindow.h"

2   #include "PfApplication.h"

3   #include <stream.h>

4

5   pfLight * PfWindow::Sun;

6   pfEarthSky * PfWindow::ESky;

7

8   // This list describes the GLX widget

9   // to be created. GLX_NOCONFIG means

10  // "give me the biggest."

11  static GLXconfig regularGlxConfig [] = {

12   GLX_NORMAL, GLX_BUFSIZE, GLX_NOCONFIG,

13   GLX_NORMAL, GLX_ZSIZE, GLX_NOCONFIG,

14   GLX_NORMAL, GLX_DOUBLE,TRUE,

15   GLX_NORMAL, GLX_RGB, TRUE,

16   GLX_NORMAL, GLX_WINDOW,GLX_NONE,

17   0, 0, 0,

18   };

19  extern PfWindow * thePfWindow = NULL;

20  extern glx_info_struct * glx_info = NULL;

21

22

23

24

25
```

```
26  // The constructor doesn't do much, just get things
27  // allocated mainly, and initialize the external global
28  // pointer to the created object.
29  PfWindow::PfWindow(char * name) : MenuWindow(name)
30  {
31    chanNear = 0.1f;
32    chanFar = 10000.0f;
33    chanFOVhorizontal = 45.0f;
34    chanFOVvertical = -1.0f;
35    thePfWindow = this;
36  }
37
38  // This function is called by the PfApplication prior
39  // to pfConfig() so forked processes can access
40  // the pfMalloc'ed data structures
41  void PfWindow::allocateSharedData()
42  {
43          arena = pfGetSharedArena();
44
45  // Place the window size in shared memory so draw process
46  // can access it
47          win_x_size = (int *)pfMalloc(sizeof(int),arena);
48          win_y_size = (int *)pfMalloc(sizeof(int),arena);
49
50  // Initialize to reasonable values
51          *win_x_size = XMAXSCREEN;
52          *win_y_size = YMAXSCREEN;
53  // Put glx_info structure in shared memory for
54  // the openGlxConnection function to do its magic
55          glx_info = (glx_info_struct *)
56                  pfMalloc(sizeof(glx_info_struct),arena);
57  }
```

```
58  // This initialization function provides a basic framework
59  // for configuring a single pipe and channel. A derived
60  // class could build on this framework by redefining an
61  // initialize() function that called the
62  // PfWindow::initialize() function, followed by
63  // additional steps such as creating a PfEarthSky, etc.
64  // The rewritten function could also change the default
65  // values of such variablesas chanNear or
66  // chanFOVhorizontal.
67  void PfWindow::initialize()
68  {
69   MenuWindow::initialize();
70   createCommandWindow();
71   pipe = pfGetPipe(0);
72   channel = pfNewChan(pipe);
73   pfChanNearFar(channel, chanNear, chanFar);
74   pfChanFOV(channel, chanFOVhorizontal, chanFOVvertical);
75   Sun = pfNewLight(NULL);
76   ESky = pfNewESky();
77   pfESkyMode (ESky, PFES_BUFFER_CLEAR, PFES_SKY_CLEAR);
78   pfChanESky(channel, ESky);
79  // Since the DrawChannel function is static, we need
80  // to pass which instance of the PfWindow class is calling
81   pfChanDrawFunc(channel, &PfWindow::DrawChannel);
82   pfChanScene(channel, thePfApplication->scene());
83  }
84  // The next two functions are defined, but empty.
85  // This gives the user the option of whether or not
86  // to implement them in his application.
87  void PfWindow::createMenuPanes ()
88  {//Empty}
```

```
89  void PfWindow::createCommandWindow()

90  {//Empty}

91

92  // This creates the GlxMDraw widget in which the

93  // Performer rendering will take place

94  Widget PfWindow::createWorkArea ( Widget w)

95  {

96   parent = w;

97   Arg wargs[10];

98

99

100  // Create the gl widget.

101  int n = 0;

102  XtSetArg(wargs[n], GlxNglxConfig,

103                          regularGlxConfig); n++;

104  glw = GlxCreateMDraw(parent, "glwidget", wargs, n);

105

106

107  // Add callbacks to the glw widget.

108  XtAddCallback(glw, GlxNginitCallback,

109              &PfWindow::redrawCB, (XtPointer) this);

110 XtAddCallback(glw, GlxNexposeCallback,

111              &PfWindow::redrawCB, (XtPointer) this);

112 XtAddCallback(glw,GlxNresizeCallback,

113              &PfWindow::redrawCB, (XtPointer) this);

114

115 // Add in a callback for processing input keys

116 // from the keyboard.

117  XtAddCallback(glw, GlxNinputCallback,

118              &PfWindow::inputCB, (XtPointer) this);

119  return glw;

120 }
```

```
121 // Another empty function that gives the designer the
122 // chance to customize the application.
123 void PfWindow::setupChannel()
124 {//Empty}
125
126
127 // This is a static callback wrapper. It checks the type
128 // of callback and invokes the proper member function
129 void PfWindow::redrawCB(Widget w,
130                             XtPointer clientdata,
131                             XtPointer calldata)
132 {
133   PfWindow *obj = (PfWindow *)clientdata;
134   GlxDrawCallbackStruct *cb = (GlxDrawCallbackStruct *)calldata;
135
136   switch (cb->reason)
137   {
138         case GlxCR_GINIT:
139          obj->init(w, cb);
140          break;
141         case GlxCR_RESIZE:
142          obj->resize(w, cb);
143          break;
144        default:
145          break;
146   }
147 }
```

```
148 // This member function is called at widget realization
149 // it holds the key to switching control of the GL context
150 // to the draw process
151 void PfWindow::init(Widget w, GlxDrawCallbackStruct * cb)
152 {
153  *win_x_size = cb->width;
154  *win_y_size = cb->height;
155  // Disconnect from the GL widget and allow the draw
156  // process to connect to it
157          Display * display = XtDisplay(w);
158          Window xWindow = XtWindow(w);
159
160 // Place info in shared memory so draw process can attach
161 // to GLXwidget.
162          glx_info->display_name = XDisplayName(NULL);
163          glx_info->xWindow = xWindow;
164
165 // Release exclusive hold on GLXwidget.
166          GLXunlink(display, xWindow);
167
168 // Performer will now call openGLXconnection in the
169 // draw process.
170          pfInitPipe(pipe, &PfWindow::OpenGlxConnection);
171 }
172
173 // Called anytime a resize event occurs. Resets the
174 // window parameters
175 void PfWindow::resize(Widget, GlxDrawCallbackStruct * cb)
176 {
177  *win_x_size = cb->width;
178  *win_y_size = cb->height;
179 }
```

```
180 // A simple draw function, this will most likely
181 // be redefined by derived classes
182 void PfWindow::DrawChannel(pfChannel * channel, void *)
183 {
184  pfLightOn(Sun);
185  pfClearChan(channel);
186  pfDraw();
187 }
188 // Static wrapper for the member function
189 void PfWindow::OpenGlxConnection(pfPipe * p)
190 {
191 // invoke the openGlxConnection this way so that
192 // derived classes can expand on the function. We use a
193 // global object pointer since there is no way to pass
194 // user data to this function
195
196         thePfWindow->openGlxConnection(p);
197 }
198
199 // Called by the draw process. Allows the draw process
200 // to get the necessary information about the GlxMDraw
201 // widget and attach to the GL context.
202 void PfWindow::openGlxConnection(pfPipe * p)
203 {
204  Display * display =
205                     XOpenDisplay(glx_info->display_name);
206  Window glx_window = glx_info->xWindow;
207
208  XWindowAttributes attributes;
209  XGetWindowAttributes(display, glx_window, & attributes);
210  int screenNo = XScreenNumberOfScreen(attributes.screen);
211
```

```
212 // Use the same configuration here that was used
213 // in creating the widget.
214 GLXconfig * config;
215 config = GLXgetconfig(display,
216                             screenNo, regularGlxConfig);
217 if (config == 0)
218 {
219         fprintf(stderr,
220             "No visual found to match request in GLXgetconfig\n");
221         exit(1);
222 }
223 // Find the window entry and set it to have the same
224 // window id of the GLXwidget's window.
225 for (int i = 0; config[i].buffer; i++)
226         if (config[i].buffer == GLX_NORMAL &&
227          config[i].mode == GLX_WINDOW)
228         {
229          config[i].arg = (int)glx_window;
230         }
231
232 // Connect the GL context to the GLXwidget created by the
233 // application process.
234 if (GLXlink(display, config))
235 {
236         fprintf(stderr, "Error in GLXlink\n");
237         exit(1);
238 }
239 // Make it the current window.
240 GLXwinset(display, glx_window);
241
242 zbuffer(TRUE);
243 }
```

```
244 // Callback wrapper for member function
245 void   PfWindow::inputCB(Widget,   XtPointer   clientdata,   XtPointer
calldata)
246 {
247         PfWindow * obj = (PfWindow *)clientdata;
248         obj->input(calldata);
249 }
250
251 // Performer calls these GL routines every frame.
252 // They are very slow because they require a round trip
253 // to the X server. These are much faster replacements.
254
255 extern "C" void
256 getorigin(long * x, long * y)
257 {
258  *x = 0;
259  *y = 0;
260 }
261
262 extern "C" void
263 getsize(long * x, long * y)
264 {
265  *x = *win_x_size;
266  *y = *win_y_size;
267 }
```

LIST OF REFERENCES

[BHAT78]   Bhattacharyya, Rameswar, _Dynamics of Marine Vehicles_, John Wiley & Sons, New York, 1978.

[BLAG62]   Blagoveshchensky, S.N., _Theory of Ship Motions_, Dover Publications, Inc., New York, 1962.

[BRUT92]   Brutzman, Donald P., et al, "Integrated Simulation for Rapid Development of Autonomous Underwater Vehicles," Proceedings of IEEE Oceans Engineering Society Conference AUTONOMOUS UNDERWATER VEHICLES 92, 3-4 JUN 92

[CORB93]   Corbin, Daniel P., "NPSNET: Environmental Effects for a Real-Time Virtual World Battlefield Simulator," Master's Thesis, Naval Postgraduate School, Monterey CA, September 1993.

[COVI92]   Covington, James H., Observations based on personal experience gained while serving aboard U.S.S. Will Rogers (SSBN 659), April 1989 through February 1992.

[DEBE57]   De Beurs, J., _Speed and Pitching - A Preliminary Study of the Pitching Motion and a Critical Study of the Trochoidal Wave Theory_, N.V. Uitgeverij v.h. A. Kemperman, Haarlem, 1957.

[EARL84]   Earle, Marshall D., et al, _A Practical Guide to Ocean Wave Measurement and Analysis_, ENDECO, Inc., Marion, MA 1984.

[FOUR86]   Fournier, A., "A Simple Model of Ocean Waves," _Computer Graphics_, 20(4), 75-84, (Proc. SIGGRAPH '86).

[HALL81]   Halliday, David et al., _Fundamentals of Physics_, Second Edition, John Wiley and Sons, New York, 1981.

[HEAL93]   Healey, Anthony J., "Dynamics of Marine Vehicles," Naval Postgraduate School, Monterey CA, 1993.

[HELL91]   Heller, Dan, _Motif Programming Manual_, Volume 6, O'Reilly and Associates, Sebastopol, CA 1991.

[MACK91]   Mackey, Randall L., "NPSNET: Hierarchical Data Structures For Real-Time Three-Dimensional Visual Simulation," Master's Thesis, Naval Postgraduate School, Monterey CA, September 1991

[McMI92]   McMinds, Donald L., _Mastering OSF/Motif Widgets_, Addison-Wesley, Reading MA, 1992..

[MAX81]   Max, N., "Vectorized Procedural Models for Natural Terrain: Waves and Islands in The Sunset," _Computer Graphics_, 15(3), 314-17, (Proc. SIGGRAPH '81).

[NEWM77]   Newman, J.N., _Marine Hydrodynamics_, The MIT Press, Cambridge, Mass., 1977.

[NYE90]   Nye, Adrian and O'Reilly, Tim, _X Toolkit Intrinsics Programming Manual_, Volume Four, O'Reilly and Associates, Inc., Sebastopol, CA 1990.

[OLSE93]   Olsen, Wade, "pfMotif" (source code and comments), Silicon Graphics Computer Systems, Mountain View CA, 1993.

[PEAC86]   Peachey, D.R., "Modeling Waves and Surf," *Computer Graphics*, 20(4), 65-74, (Proc. SIGGRAPH '86).

[PERL85]   Perlin, K., "An Image Synthesizer," *Computer Graphics*, 19(3), 287-96, (Proc. SIGGRAPH '85).

[PRAT93]   Pratt, David R., "A Software Architecture for the Construction and Management of Real-Time Virtual Worlds," Doctoral Dissertation, Naval Postgraduate School, Monterey CA, June 1993.

[PRES88]   Press, William H., et. al., Numerical Recipes in C : The Art of Scientific Computing, Cambridge University Press, New York, 1988.

[SCHN92]   Schneiderman, Ben, Designing The User Interface: Strategies For Effective Human-Computer Interaction, Addison-Wesley, Reading MA 1992

[SGI92]   Silicon Graphics, Inc., IRIS Performer Programming Guide, Document Number 007-1680-010, Mountain View, CA, 1992.

[SGI92a]   Silicon Graphics, Inc., IRIS Performer Man Pages, Document Number 007-1681-010, Mountain View, CA, 1992.

[SGI93]   Silicon Graphics, Inc., IRIS Sgm Man Pages, Mountain View, CA, 1993.

[YOUN92]   Young, Douglas A., Object-Oriented Programming with C++ and OSF/Motif, Prentice Hall, Englewood Cliffs, NJ 1992.

[YOUN93]   Young, Roy D., "NPSNET-IV: A Real-Time, 3D Distributed, Interactive Virtual World," Master's Thesis, Naval Postgraduate School, Monterey CA, September 1993.

[ZESW93]   Zeswitz, Stephen R., "NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange," Master's Thesis, Naval Postgraduate School, Monterey CA, September 1993.

[ZYDA93]   Zyda, Michael, et al, "NPSNET and the Naval Postgraduate School Graphics and Video Laboratory," Presence Vol. 2, No. 3.

INITIAL DISTRIBUTION LIST